Empirical Software Engineering (2020) 25:434–487
https://doi.org/10.1007/s10664-019-09778-7

Selecting fault revealing mutants

Thierry Titcheu Chekam[1] · Mike Papadakis[1] · Tegawendé F. Bissyandé[1] · Yves Le Traon[1] · Koushik Sen[2]

Published online: 18 December 2019

Abstract
Mutant selection refers to the problem of choosing, among a large number of mutants, the (few) ones that should be used by the testers. In view of this, we investigate the problem of selecting the fault revealing mutants, i.e., the mutants that are killable and lead to test cases that uncover unknown program faults. We formulate two variants of this problem: the fault revealing mutant selection and the fault revealing mutant prioritization. We argue and show that these problems can be tackled through a set of 'static' program features and propose a machine learning approach, named *FaRM*, that learns to select and rank killable and fault revealing mutants. Experimental results involving 1,692 real faults show the practical benefits of our approach in both examined problems. Our results show that *FaRM* achieves a good trade-off between application cost and effectiveness (measured in terms of faults revealed). We also show that *FaRM* outperforms all the existing mutant selection methods, i.e., the random mutant sampling, the selective mutation and defect prediction (mutating the code areas pointed by defect prediction). In particular, our results show that with respect to mutant selection, our approach reveals 23% to 34% more faults than any of the baseline methods, while, with respect to mutant prioritization, it achieves higher average percentage of revealed faults with a median difference between 4% and 9% (from the random mutant orderings).

Keywords Mutation testing · Machine learning · Mutant selection · Mutant prioritization

1 Introduction

Mutation testing has been shown to be one of the most effective techniques with respect to fault revelation (Titcheu Chekam et al. 2017). Researchers typically use mutation as an assessment mechanism (measuring effectiveness) for their techniques (Papadakis et al. 2018a), but it can be used as every other test criterion. To this end, mutation can be used

Communicated by: Jeff Offutt

✉ Thierry Titcheu Chekam
thierry.titcheu-chekam@uni.lu

Extended author information available on the last page of the article.

to assess the effectiveness of test suites or to guide test generation (Ammann and Offutt 2008; Fraser and Zeller 2012; Petrovic and Ivankovic 2018; Papadakis et al. 2018b; Titcheu Chekam et al. 2017).

Unfortunately, mutation testing is expensive. This is due to the large number of mutants that require analysis. An important cost parameter is the so-called *equivalent mutants*, which are mutants forming equivalent program versions (Papadakis et al. 2015; Ammann and Offutt 2008). These need to be manually inspected by testers since their automatic identification is not always possible (Budd and Angluin 1982).

While the problem of the equivalent mutants have been partly addressed by recent methods such as the Trivial Compiler Equivalence (TCE) (Papadakis et al. 2015), the problem of the large number of mutants remains challenging. Yet, addressing this problem will in return contribute to addressing the equivalent mutant problem: any approach that is effective in reducing the large number of mutants, would indirectly reduce the equivalent mutant problem since less equivalent mutants will be available.

Nevertheless, producing a large number of mutants is impractical. The mutants need to be analyzed, compiled, executed and killed by test cases. Perhaps, more importantly testers need to manually analyse them in order to design effective test cases. The scalability, or lack thereof, of mutation testing, with respect to the number of mutants to be processed, is thus a key factor that hinders its wide applicability and large adoption (Papadakis et al. 2018a). Consequently, if we can find a lightweight and reasonably effective way to diminish the number of mutants without sacrificing the power of the method, we would then manage to significantly improve the scalability of the method. Since the early days of mutation testing, researchers attempted to find such solutions by forming many mutant reduction strategies (Papadakis et al. 2018a), such as selective mutation (Offutt et al. 1993; Wong and Mathur 1995a) and random mutant selection (T Acree et al. 1979).

Our goal is to form a mutant selection technique that identifies killable mutants that are fault revealing, prior to any mutant execution. We consider as fault revealing, any mutant (i.e. test objective) that leads to test cases capable of revealing the faults in the program under test. We argue that such mutants are program specific and can be identified by a set of static program features. In this respect, we need features that are simultaneously generic, in order to be widely applicable, and powerful to approximate well the program and mutant semantics.

We advance in this research direction by proposing a machine learning-based approach, named *FaRM* , which learns on code and mutants' properties, such as mutant type and mutation location in program control-flow graphs, as well as code complexity and program control and data dependencies, to (statically) classify mutants as likely killable/equivalent and likely fault revealing. This approach is inspired by the prediction modelling line of research, which has recorded high performance by using machine learning to triage likely error-prone characteristics of code (Menzies et al. 2007; Kamei and Shihab 2016).

The use case scenario of *FaRM* is a standard testing scenario where mutants are used as test objectives, guiding test generation. To achieve this, we train on a set of faulty programs that have been tested with mutation testing, prior to any testing or test case design for the particular system under analysis. Then, we predict the killable and fault revealing mutants based on which we test the particular system under analysis. The training corpus can include previously developed projects (related to the targeted application domain) or previous releases of the tested software. In a sense, we train on system(s), say x, and select mutants on the system under test, say y, where $x \neq y$.

Experimental results using 10-Fold cross validation on 1,692 + 45 faulty program versions show a high performance of *FaRM* in yielding an adequately selected set of mutants.

In particular our method achieves statistically significantly better results than the random, selective mutation and defect prediction (mutating the areas predicted by defect prediction), mutant selection baselines by revealing 23% to 34% more faults than any of the baselines. Similarly, our mutant prioritization method achieves statistically significant higher Average Percentage of Faults Detected (APFD) (Henard et al. 2016) values than the random prioritisation (4% to 9% higher in the median case). With respect to test execution, we show that our selection method requires less execution time (than random).

We also demonstrate that our method is capable of selecting killable (non-equivalent) mutants. In particular, by building an equivalent classification method, using our features, we achieve an AUC value of 0.88 and 95%, 35% precision and Recall. These results indicate drastic reductions on the efforts required by the analysis of equivalent mutants. A combined approach, named *FaRM** , achieves similar to *FaRM* fault revelation, but potentially at a lower cost (lower number of equivalent mutants), indicating the capabilities of our method.

In summary, our paper makes the following contributions:

- It introduces the fault revealing mutant selection and fault revealing mutant prioritization problems.
- It demonstrates that the killability and fault revealing utility of mutants can be captured by simple static source code metrics.
- It presents *FaRM* , a mutant selection technique that learns to select and rank mutants using standard machine learning techniques and source code metrics.
- It provides empirical evidence suggesting that *FaRM* outperforms the current state-of-the-art mutant selection and mutant prioritization methods by revealing 23% to 34% more faults and achieving 4% to 9% higher average percentage of revealed faults, respectively.
- It provides a publicly available dataset of feature metrics, kill and fault revelation matrices that can support reproducibility, replication and future research.

The paper is organized as follows. Section 2 provides background information on mutation testing, the mutant selection problem and defines the targeted problem(s). Section 3 overviews the proposed approach. Evaluation research questions are enumerated in Section 4, while experimental setup is described in Section 5 and experimental results are presented in Section 6. A detailed discussion on the applicability of our approach and the threats to validity are given in Section 7, and related work is discussed in Section 8. Section 9 concludes this work.

2 Context

2.1 Mutation Testing

Mutation testing (DeMillo et al. 1978) is a test adequacy criterion that sets the revelation of artificial defects, called mutants, as the requirements of testing. As every test criterion, mutation assists the testing process by defining test requirement that should be fulfilled by the designed test cases, i.e., defining when to stop testing.

Software testing research has shown that designing tests that are capable of revealing mutant-faults results in strong test suites that in turn reveal real faults (Frankl et al. 1997; Li et al. 2009; Titcheu Chekam et al. 2017; Papadakis et al. 2018a; Just et al. 2014b) and are capable of subsuming or almost subsuming all other structural testing criteria (Offutt et al. 1996b; Frankl et al. 1997; Ammann and Offutt 2008).

Mutants form artificially-generated defects that are introduced by making changes to the program syntax. The changes are introduced based on specific syntactic transformation rules, called *mutation operators*. The syntactically changed program versions form the mutant-faults and pose the requirement of distinguishing their observable behaviour from that of the original program. A mutant is said to be *killed*, if its execution distinguishes it from the original program. In the opposite case it is said to be *alive*.

Mutation quantifies test thoroughness, or test adequacy (DeMillo et al. 1978, 1991; Frankl and Iakounenko 1998), by measuring the number of mutants killed by the candidate test suites. In particular, given a set of mutants, the ratio of those that are killed by a test suite is called mutation score. Although all mutants differ syntactically from the original program, they do not always differ semantically. This means that there are some mutants that are semantically equivalent to the original program, while being syntactically different (Offutt and Craft 1994; Papadakis et al. 2015). These mutants are called equivalent mutants (DeMillo et al. 1978; Offutt and Craft 1994) and have to be removed from the test requirement set.

Mutation score denotes the degree of achievement of the mutation testing requirements (Ammann and Offutt 2008). Intuitively, the score measures the confidence on the test suites (in the sense that mutation score reflects the fault revelation ability). Unfortunately, previous research has shown that the relation between killed mutants and fault revelation is not linear (Frankl et al. 1997; Titcheu Chekam et al. 2017) as fault revelation improves significantly only when test suites reach high mutation score levels.

2.2 Problem Definition

Our goal is to select among the many mutants the (few) ones that are fault revealing, i.e., mutants that lead to test cases that reveal existing, but unknown, faults. This is a challenging goal since only 2% (according to our data) of the killable mutants are fault revealing.

The fault revealing mutant selection goal is different from that of the "traditional" mutant reduction techniques, which is to reduce the number of mutants (Offutt et al. 1996a; Wong and Mathur 1995b; Ferrari et al. 2018; Papadakis et al. 2018a). Mutant reduction strategies focus on selecting a small set of mutants that is representative of the larger set. This means, that every test suite that kills the mutants of the smaller set, also kills the mutants of the large set. Figure 1 illustrates our goal and contrasts it with the "traditional" mutant reduction problem. The blue (and smallest) rectangle on the figure represents the targeted output for the fault revealing mutant selection problem.

Previous research (Papadakis et al. 2018b, c) has shown that the majority of the mutants, even in the best case, are "irrelevant" to the sought faults. This means that testers need to analyse a large number of mutants before they can find the actually useful ones (the fault revealing ones), wasting time and effort. According to our data, 17% of the minimal mutants (ideal mutant reduction), i.e., subsuming mutants (a set of mutants with minimal overlap that are sufficient for preserving test effectiveness Jia and Harman 2009; Kintis et al. 2010; Ammann et al. 2014) is fault revealing. This also indicates that the majority of the mutants, even in the best case, are "irrelevant" to the sought faults. We therefore claim that mutation testing should be performed only with the mutants that are most likely to be fault revealing. This will make possible the best effort application of the method.

Formally, we consider two aspects of this selection problem: the mutant selection one and the mutant prioritization one.

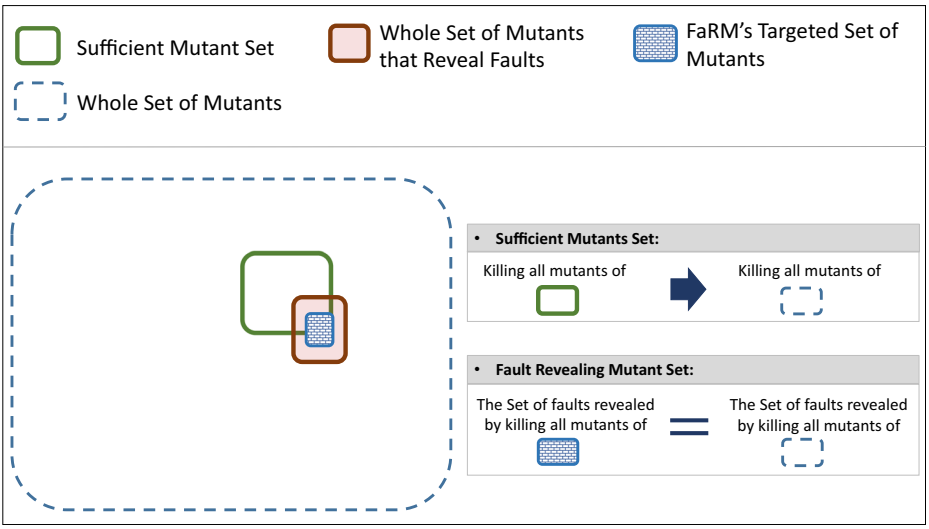


Fig. 1 Fault revealing mutant selection. Contrast between sufficient mutant set selection and fault revealing mutant selection. Sufficient mutant set selection aims at selecting a minimal subset of mutants that is killed by tests that also kill the whole set of mutants. Fault revealing mutant selection aims at selecting a minimal subset of mutants that is killed by tests that reveal the same underlying faults as the tests that kill the whole set of mutants

The **fault revealing mutant selection problem** is defined as:

Given: A set of mutants M for program P.
Problem: Subset selection. Select a subset of mutants, $S \in M$, such that $F(S) = F(M)$ and $(\forall m \in S), (F(S - \{m\}) \neq F(M))$.

S represents a subset of M; $F(X)$ represents the number of faults in P that are revealed by the test suites that kill all the mutants of the set X. In practice, the challenge is to approximate well S, statically and prior to any test execution, by finding a relatively good trade-off between the number of selected mutants (to minimise) and the number of faults revealed by their killing (to maximize).

Similarly, the **fault revealing mutant prioritization problem** is defined as:

Given: A set of mutants, M and the set of permutations of M, PM for program P.
Problem: Find $Pm' \in PM$ such that $(\forall Pm'')(Pm'' \in PM)\ (Pm'' \neq Pm')\ [f(Pm') \geq f(Pm''))]$

PM represents the set of all possible mutant orderings of M, and $f(X)$ represents the average percentage of faults revealed by the test cases that kill the selected mutants in the given order X (measures the area under the curve representing the faults revealed by the killing of each one of the mutants in the order). The challenge is to statically and prior to any test execution, rank the mutants so that the fault revealing potential is maximized

when killing any (arbitrary) number of them. The idea is that fault revelation is maximized whenever the tester decides to stop killing mutants.

2.3 Mutant Selection

In the literature many mutant selection methods have been proposed (Papadakis et al. 2018a; Ferrari et al. 2018) by restricting the considered mutants according to their types, i.e., applying one or more mutant operators. Empirical studies (Kurtz et al. 2016; Deng et al. 2013), have shown that the most successful strategies are the statement deletion (Deng et al. 2013) and the E-Selective mutant set (Offutt et al. 1993, 1996a). We therefore compare our approach with these methods. We also consider the random mutant selection (T Acree et al. 1979) since there is evidence demonstrating that it is particularly effective (Zhang et al. 2010; Papadakis and Malevris 2010b).

2.3.1 Random Mutant Selection

Random mutant sampling (T Acree et al. 1979) forms the simplest mutant selection technique, which can be considered as a natural baseline method. Interestingly, previous studies found it particularly effective (Zhang et al. 2010; Papadakis and Malevris 2010b). Therefore, we compare with it.

We use two random selection techniques, named as SpreadRandom and DummyRandom. SpreadRandom iteratively goes through all program statements (in random order) and selects mutants (one mutant among the mutants of each statement), while DummyRandom selects them from the set of all possible mutants. The first approach is expected to select mutants residing on most of the program statements, while the second one is expected to make a uniform selection.

2.3.2 Statement Deletion Mutant Selection

Mutant selection based on statement deletion is a simple approach that, as the name suggests, deletes every program statement (once at a time). To avoid introducing compilation issues (mutants that do not compile) and introduce relatively strong mutants, the statement deletion is usually applied on parts of a statement (deleting parts of expressions, i.e., the expression $a + b$ becomes a or b). Empirical studies have shown that statement deletion mutant selection is powerful (achieves a very good trade-off between the number of selected mutants and test effectiveness) and has the advantage of introducing few equivalent mutants (Deng et al. 2013).

2.3.3 E-selective Mutant Selection

E-Selective refers to the 5 operator mutant set introduced by Offutt et al. (1993, 1996a). This set is the most popular operator set (Papadakis et al. 2018a) that is included in most of the modern mutation testing tools. This set includes the mutants related to relational, logical (including conditional), arithmetic, unary and absolute mutations. According to the study of Offutt et al. (1996a) this set has the same strengths as a much larger comprehensive set of operators. Although there is empirical evidence demonstrating that the E-Selective set has lower strengths than a more comprehensive set of operators (Kurtz et al. 2016), it still provides a very good trade-off beetween selected mutants and strengths (Kurtz et al. 2016).

2.4 Mutant Prioritization

Mutant prioritization has received little or even no attention in literature (refer to the Related Work Section 8 for details). Given the absence of other methods, we compare our approach with the random baselines. We also consider alternative schemes, such as Defect Prediction prioritization.

2.4.1 Random Mutant Prioritization

Random mutant prioritization forms a natural baseline for our approach. Comparing with random orderings is a common practice in test case prioritization studies (Rothermel et al. 2001; Henard et al. 2016) and shows the ability of the prioritization method to systematically order the sought elements. Similarly to mutants selection, we applied two random ordering techniques, the SpreadRandom and DummyRandom. SpreadRandom orders mutants by iteratively going through all program statements (in random order) and selects one mutant among the mutants of each statement (statement-based orders), while DummyRandom orders them from the mutant set (uniform orders).

2.4.2 Defect Prediction Mutant Prioritization

Naturally, one of the main attributes determining the utility of the mutants is their location. Thus, instead of selecting mutants based on other properties, one could select them based on their location. To this end, we form a prioritization method that predicts and orders the error-prone code locations, i.e., code parts that are most likely to be faulty. Then, we mutate the predicted code areas and form a baseline method. Such an approach is in sense equivalent to the application of mutation testing on the results of defect prediction. Moreover, such a comparison demonstrates that mutants depend on the attributes (features) we train on not solely on their location.

3 Approach

Our objective is to select mutants that lead to effective test cases. In view of this, we aim at selecting and prioritizing mutants so that we reveal most of the faults by analysing the smallest possible number of mutants.

We conjecture that mutant selection strategies should account for the properties that make them killable and fault revealing. Defect prediction studies (Menzies et al. 2007; Kamei and Shihab 2016) investigated properties related to error-prone code locations, but not related to mutants. Mutation testing is a behaviour oriented criterion and requires mutants introducing small and useful semantic deviations. Therefore, we propose building a model, which captures the essential properties that make mutants valuable (in terms of their utility to reveal faults).

Figure 2 depicts the *FaRM* approach, which learns to rank mutants according to their fault revealing potential (likelihood to reveal (unknown) faults). Initially, *FaRM* applies supervised learning on the mutants generated from a corpus of faulty program versions, and builds a prediction model. This model is then used to predict the mutants that should be used to test the particular instance of the program under test. This means that at the time of testing and prior to any mutant execution, testers can use and focus only on the most important mutants.

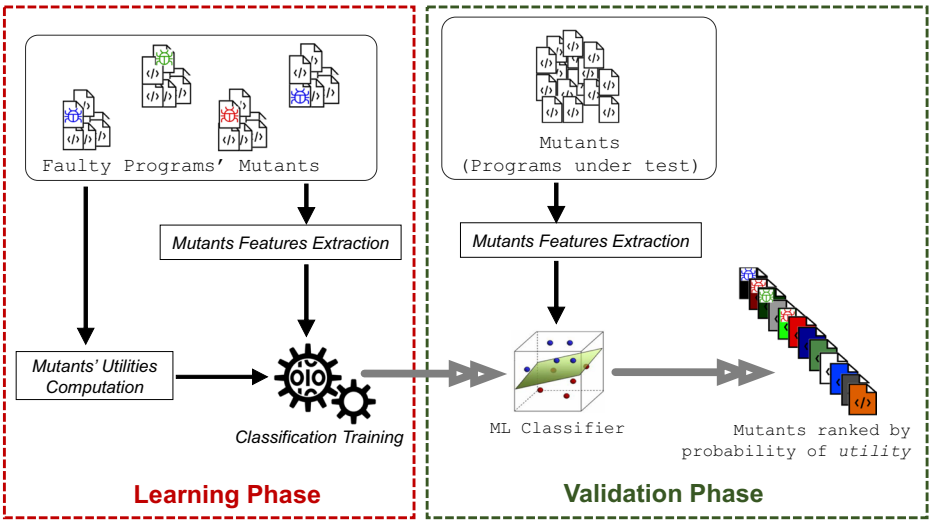


Fig. 2 Overview of the *FaRM* approach. Initially, *FaRM* applies supervised learning on the mutants generated from a corpus of faulty program versions, and builds a prediction model that learns the fault revealing mutant characteristics. This model is then used to predict the mutants that should be used to test other program versions. This means that at the time of testing and prior to any mutant execution, testers can use and focus only on the most important mutants

Regarding *FaRM* supervised learning training phase (when the prediction model is built), the faulty programs mutants's features are extracted and used as training data's features and, their utilities are computed and used as training data's expected output. The mutant's utility for fault revealing and killable mutant prediction is respectively the mutants' fault revealing and killability information. Regarding the validation phase, features of the mutants of the program under test are extracted and used as validation data's features to predict the mutants' utilities with the trained model. Mutants with high predicted utility are the useful ones.

Definition For a given problem, we define as classifier's *performance* the prediction performance of the classifier, which is the accuracy of the predictions (precision, recall, F-measure and Area Under Curve metrics that are detailed in Section 5.3) of the classifier for the given problem.

ML-based measurement of mutant utility. The selection process in *FaRM* is based on training a predictor for assessing the probability of a mutant to reveal faults. To that end, we explore the capability of several features, which are designed to reflect specific code properties which may discriminate a useful mutant from another. Let us consider a mutant M associated to a code statement S_M on which the mutation was applied. This mutant can be characterized from various perspectives with respect to (1) the complexity of the relevant mutated statement, (2) the position of the mutated code in the control-flow graph, (3) dependencies with other mutants, (4) the nature of the code block where S_M is located.

ML features for characterizing mutants. Recently, the studies of Wen et al. (2018), Just et al. (2017), and Petrovic and Ivankovic (2018) found a strong connection between mutants'

utility and the surrounding code (captured by the AST father and child nodes). Therefore, in addition to the mutant types, typically considered by selective mutation approaches (Offutt et al. 1996a; Namin et al. 2008; Papadakis et al. 2018a), we also considered the information encoded on the program AST. We include three such features, the Data type at the mutant location, the parent AST node (of the mutant expression) and the child AST node (of the mutant expression), in our machine learning classification scheme.

Let B_M be the control-flow graph (CFG) basic block associated to a mutated statement S_M containing the mutated expression E_M. Table 1 provides the list of all 28

Table 1 Description of the static code features

Complexity	Complexity of statement S_M approximated by the total number of mutants on S_M
CfgDepth	Depth of B_M according to CFG
CfgPredNum	Number of predecessor basic blocks, according to CFG, of B_M
CfgSuccNum	Number of successors basic blocks, according to CFG, of B_M
AstNumParents	Number of AST parents of E_M
NumOutDataDeps	Number of mutants on expressions data-dependents on E_M
NumInDataDeps	Number of mutants on expressions on which E_M is data-dependent
NumOutCtrlDeps	Number of mutants on statements control-dependents on E_M
NumInCtrlDeps	Number of mutants on expressions on which E_M is control-dependent
NumTieDeps	Number of mutants on E_M
AstParentsNumOutDataDeps	Number of mutants on expressions data-dependent onE_M's AST parent statement
AstParentsNumInDataDeps	Number of mutants on expressions on which E_M's AST parent expression is data-dependent
AstParentsNumOutCtrlDeps	Number of mutants on statements control-dependent on E_M's AST parent expression
AstParentsNumInCtrlDeps	Number of mutants on expressions on which E_M's AST parent expression is control-dependent
AstParentsNumTieDeps	Number of mutants on E_M's AST parent expression
TypeAstParent	Expression type of AST parent expressions of E_M
TypeMutant	Mutant type of M as matched code pattern and replacement. Ex: $a + b \rightarrow a - b$
TypeStmtBB	CFG basic block type of B_M. Ex: $if - then,\ if - else$
AstParentMutantType	Mutant types of M's AST parents
OutDataDepMutantType	Mutant types of mutants on expressions data-dependents on E_M
InDataDepMutantType	Mutant types of mutants on expressions on which E_M is data-dependent
OutCtrlDepMutantType	Mutant types of mutants on statements control-dependents on E_M
InCtrlDepMutantType	Mutant types of mutants on expressions on which E_M is control-dependent
AstChildHasIdentifier	AST child of expression E_M has an identifier
AstChildHasLiteral	AST child of expression E_M has a literal
AstChildHasOperator	AST child of expression E_M has an operator
DataTypesOfOperands	Data types of operands of E_M
DataTypeOfValue	Data type of the returned value of E_M

features that we extract from each mutant. The features named *TypeAstParent*, *TypeMutant*, *TypeStmtBB*, *AstParentMutantType*, *OutDataDepMutantType*, *InDataDepMutantType*, *OutCtrlDepMutantType*, *InCtrlDepMutantType*, *DataTypesOfOperands* and *DataTypesOfValue* are categorical. We represented them using one hot encoding. Besides the categorical features listed above, all other features are numerical. The values of numerical features are normalized between 0 and 1 using *feature scaling*, more precisely *min-max normalization/scaling*.

A demonstrating example on how mutant features are computed is given in the following subsection (Section 3.2). After extracting feature values, we feed them to a machine learning classification algorithm along with the killable and fault revealing information for each mutant for a set of faults. The training process then produces two classifiers (one for the equivalent and one for the fault revealing mutants) which, given the feature values of a given mutant, they are capable of computing the utility probabilities for this mutant, i.e., its probability to be killable and its probability to be fault revealing.

By using these two classifiers we form three approaches, two of them using each one of the classifiers alone and one of them by combining them. The first two, named *FaRM* and *PredKillable* , respectively classify mutants according to their probability to be fault revealing and killable. The third one, named *FaRM** , divides the mutant set in two subsets, likely killable and likely equivalent (based on *PredKillable* predictions), separately ranks them according to their fault revealing probability and concatenates them by putting the likely killable subset first. Figure 3 show an example of mutant ranking by *FaRM**. The motivation for *FaRM** results from the hypothesis that equivalent mutants

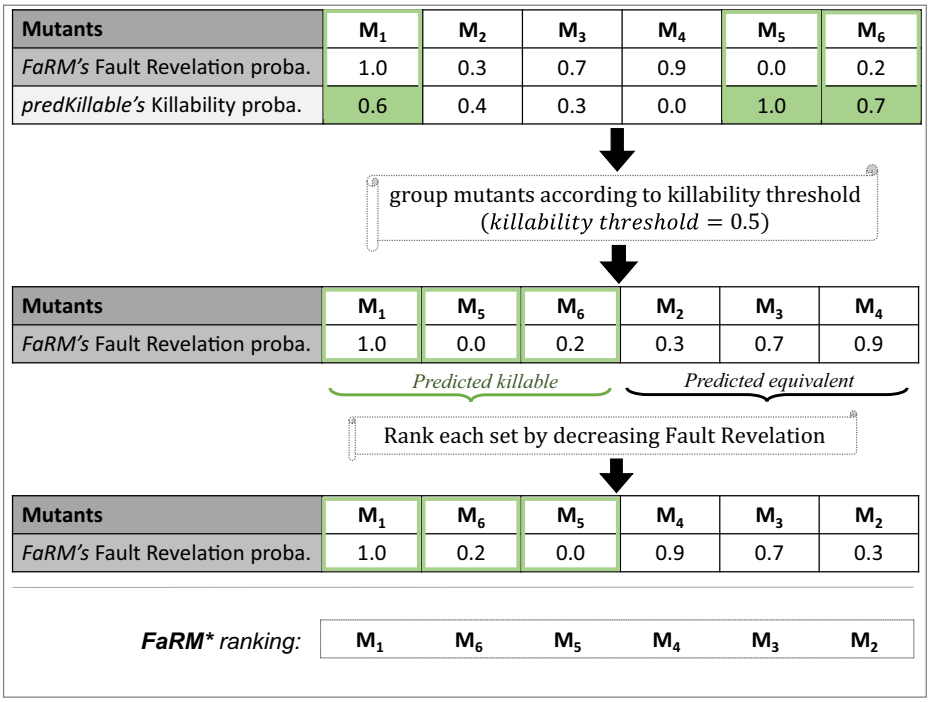


Fig. 3 Example of mutant ranking procedure by *FaRM** . the ranking is a concatenation of the ranked *predicted killable* mutants and the ranked *predicted equivalent* mutants

could be noise to *FaRM* and, *PredKillable* performs better at filtering equivalent mutants (or predicting killable mutants). Given that fault revealing mutants are killable, we expect them to have higher predicted utility value with both *FaRM* and *PredKillable*. Therefore, *FaRM** gives priority to the most likely fault revealing mutants that are also most likely killable.

We implement a prioritization scheme by considering the ranking of all mutants in accordance to the values of the developed probability measure. This forms our mutant prioritization approaches. Our mutant selection strategy sets a threshold probability value (e.g., 0.5) or a cut-off point according to the number of the top ranked mutants to keep only mutants with higher utility probability scores in the selected set. This forms our mutant selection approach. For the combined approach (*FaRM**) we divide the mutant set in the killable and equivalent subsets by using a cut-off point of 0.5.

3.1 Implementation

We implemented *FaRM* as a collection of tools in C++. We leverage stochastic gradient boosting (Friedman 2002) (decision trees) to perform supervised learning. Gradient boosting is a powerful ensemble learning technique which combines several trained weak models to perform classification. Unlike common ensemble techniques, such as random forests (Breiman 2001), that simply average models in the ensemble, boosting methods follow a constructive strategy of ensemble formation where models are added to the ensemble sequentially. At each particular iteration, a new weak, base-learner model is trained with respect to the error of the whole ensemble learnt so far (Natekin and Knoll 2013). We use the FastBDT (Keck T 2016) implementation by setting the number of trees to 1,000 and the trees depth to 5.

3.2 Demonstrating Example

Here we provide an example on how the features of Table 1 are computed. We consider the program in Fig. 4 (extracted from the Codeflaws benchmark, ID: 598-B-bug-17392756-17392766), on which mutation is applied. We present the feature extraction for a mutant M, which is created by replacing the right side decrement operator by the right side increment operator on line 16 ($m - -$ becomes $m + +$). We also present in Fig. 5a the mutant, the abstract syntax tree (AST) of the mutated statement (*while* condition) in Fig. 5b and c the control flow graph (CFG) of the function containing the mutated statement.

The features, for mutant M, are computed as following:

- The *complexity* feature value is the number of mutants generated on the statement containing the mutant M (Line 16). In this case 72 mutants. Thus, the *complexity* is 72.
- The *CfgDepth* feature value is the minimum number of basic blocks to follow, along the CFG, from *main* function's entry point to the basic block containing M (*BB2*). In this case 1 basic block as shown in Fig. 5c. Thus, the *CfgDepth* is 1.
- The *CfgPredNum* feature value is the number of basic blocks directly preceding the basic block containing M (*BB2*) on the control flow graph. In Fig. 5c there are 2 basic blocks (*BB1* and *BB3*). Thus, the *CfgPredNum* is 2.
- The *CfgSuccNum* feature value is the number of basic blocks directly following the basic block containing M (*BB2*) on the control flow graph. In Fig. 5c there are 2 basic blocks (*BB3* and *BB4*). Thus, the *CfgSuccNum* is 2.

```c
#include <stdio.h>
#include <string.h>

void rotate(char *s, int n, int k) {
    char t[10000];
    memcpy(t, s + n - k, k);          // 49 mutants
    memcpy(t + k, s, n - k);          // 65 mutants
    memcpy(s, t, n);                  // 10 mutants
}

int main(int argc, char *argv[]) {
    char s[10000];
    int m, l, r, k;
    scanf("%s", s);                   // 6 mutants
    scanf("%d", &m);                  // 3 mutants
    while (m-- > 0) {                 // 72 mutants
        scanf("%d%d%d", &l, &r, &k); // 6 mutants
        l--;                          // 55 mutants
        rotate(s + l, r - l, k);      // 60 mutants
    }
    printf("%s\n", s);                // 7 mutants
    return 0;                         // 3 mutants
}
```

Fig. 4 Example program where mutation is applied. The C language comments on each line show the number of mutants generated on the line

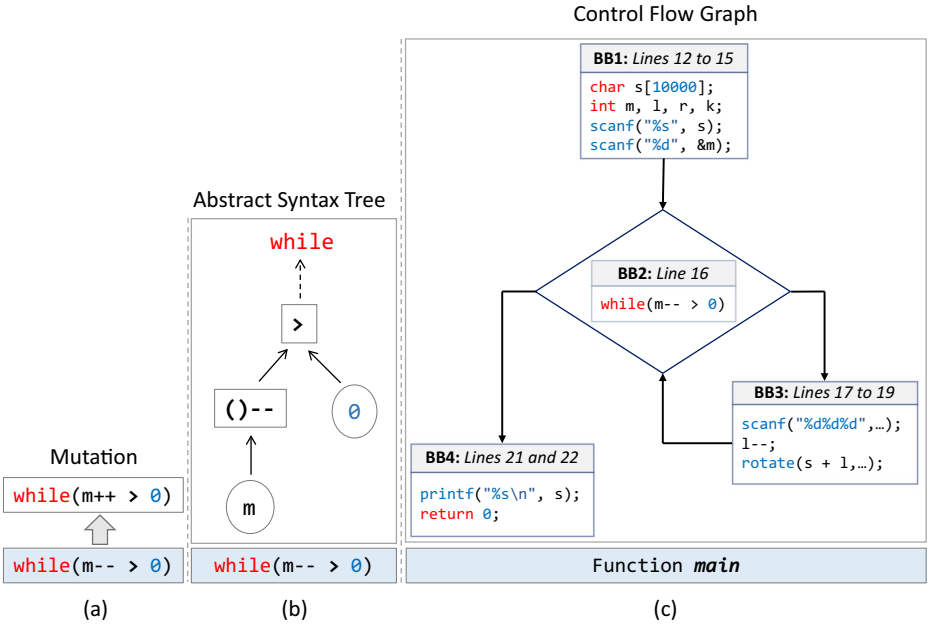


Fig. 5 **a** An example of mutant *M* from the example program from Fig. 4, **b** the abstract syntax tree of the mutated statement and **c** the control flow graph of the function containing the mutated statement

- The *AstNumParents* feature value is the number of AST parents of the mutated expression. In this case, the only AST parent is the relational expression, in Fig. 5b, whose sub-tree is rooted on the greater than sign (>). Thus the feature value is 1.
- The *NumOutDataDeps* feature value is the number of mutants on expressions data dependent on the mutated expression. In this case, looking at Fig. 4, the value of variable m written in the mutated expression $m - -$ is only used in the same expression. Thus the feature value is the number of mutants on the mutated expression $m - -$.
- The *NumInDataDeps* feature value is the number of mutants on expressions on which the mutated expression is data dependent. In this case, looking at Fig. 4, the value of variable m used on the mutated expression $m - -$ is either written on the *scanf* statement at line 15, or in the same expression. Thus the feature value is the sum of the number of mutants on the statement at line 15 and the number of mutants on the mutated expression $m - -$.
- The *NumOutCtrlDeps* feature value is the number of mutants on statements control dependent on the mutated expression. In this case, looking at Fig. 4, no statement is control dependent on the mutated expression $m - -$. Thus the feature value is 0.
- The *NumInCtrlDeps* feature value is the number of mutants on expressions on which the mutated statement is control dependent. In this case, looking at Fig. 4, no expression controls the mutated expression. Thus the feature value is 0.
- The *NumTieDeps* feature value is the number of mutants on the right decrement expression (mutated expression).
- The *AstParentsNumOutDataDeps* feature value is the number of mutants on expressions data dependent on the AST parent of the mutated expression. In this case, looking at Figs. 4 and 5b, the value of the relational expression (AST parent of $m - -$) is not used in other expressions. Thus the feature value is 0.
- The *AstParentsNumInDataDeps* feature value is the number of mutants on expressions on which the AST parent of the mutated expression is data dependent. In this case, looking at Figs. 4 and 5b, the value of the relational expression (AST parent of $m - -$) only depends on the value of expression $m - -$. Thus the feature value is the number of mutants on expression $m - -$.
- The *AstParentsNumOutCtrlDeps* feature value is the number of mutants on statements control dependent on the AST parent of the mutated expression. In this case, looking at Figs. 4 and 5b, all the statements in basic block $BB3$ are control dependent on the relational expression (AST parent of $m - -$). Thus the feature value is the sum of the number of mutants in lines 17, 18 and 19 of the code in Fig. 4.
- The *AstParentsNumInCtrlDeps* feature value is the number of mutants on expressions on which the AST parent of the mutated expression is control dependent. In this case, looking at Figs. 4 and 5b, no expression controls the relational expression (AST parent of the mutated expression $m - -$). Thus the feature value is 0.
- The *AstParentsNumTieDeps* feature value is the number of mutants on the relational expression, AST parent of the mutated right decrement expression. The feature value here is the number of mutants of the relational expression of operator greater than.
- The *TypeAstParents* feature value is AST type of the AST parent expression of the mutated expression. Here, that is the AST type of the relational expression with operator greater than.
- The *TypeMutant* feature value is the type of the mutant as a string representing the matched and replaced pattern. The feature value is "$() - - \rightarrow () + +$".

- The *TypeStmtBB* feature value is the type of the basic block containing the mutated statement. The feature value here is the type of *BB2* (see Fig. 5c), which is "While Condition".
- The *AstParentMutantType* feature value is the aggregation of types of the mutants on the AST parents of the mutated expression. That is the aggregation of the mutants types of the relational expression whose sub-tree is rooted on the greater than sign (>) as shown in Fig. 5b. The aggregation of a set of mutant types is performed by summing up the one encoding vectors of the mutants types, allowing each mutant type to be represented in the encoding.
- The *OutDataDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumOutDataDeps*.
- The *InDataDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumInDataDeps*.
- The *OutCtrlDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumOutCtrlDeps*.
- The *InCtrlDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumInCtrlDeps*.
- The *AstChildHasIdentifier* feature value is the Boolean value representing whether the mutated expression has an identifier as operand. In this case, the mutated expression has the identifier *m* as operand. Thus, the value of the feature is 1 (True).
- The *AstChildHasLiteral* feature value is the Boolean value representing whether the mutated expression has a literal as operand. In this case, the mutated expression does not have the literal as operand. Thus, the value of the feature is 0 (False).
- The *AstChildHasOperator* feature value is the Boolean value representing whether the mutated expression has an operator. In this case, the mutated expression has the operator right decrement operator −−. Thus, the value of the feature is 1 (True).
- The *DataTypesOfOperands* feature value is the datatype of the operand of the right decrement operation −−. That is the datatype of *m* which is *"int"*.
- The *DataTypeOfValue* feature value is the datatype of the value of the mutated expression, Which is *"int"* as the data type of *m*.

4 Research Questions

When building prediction methods, the first thing to investigate is their prediction ability. Thus, our first question can be stated as:

RQ1: *How well does our machine learning method predict the killable mutants?*

Similarly, our second question can be stated as:

RQ2: *How well does our machine learning method predict the fault revealing mutants?*

After demonstrating that our classification method predicts satisfactorily the fault revealing mutants, we continue by investigating its ability to practically support mutant selection with respect to the actual measure of interest, the revealed faults, and with respect to the random baseline techniques. Therefore, we investigate:

RQ3: *How do our methods compare against the random strategies with respect to the fault revealing mutant selection problem?*

In addition to the random strategies, we also compare with the current state-of-the-art mutant selection methods. Thus, we ask:

RQ4: *How do our methods compare against the E-Selection and SDL with respect to the fault revealing mutant selection problem?*

As we already discussed an alternative mutant cost reduction technique is mutant prioritization. Hence, we ask:

RQ5: *How do our methods compare against the random strategies with respect to the fault revealing mutant prioritization problem?*

In addition to the random strategies, we also compare with the defect prediction mutant prioritization baseline. Therefore, we ask:

RQ6: *How do our methods compare against the defect prediction mutant prioritization method?*

Finally, by demonstrating the benefits of our approach, we turn to investigate the generalization ability of our approach on larger and complex programs. Therefore we conclude by asking:

RQ7: *How well do our method generalise its findings on independently selected programs that are much larger and complex?*

5 Experimental Setup

5.1 Benchmarks: Programs and Fault(s)

For the purposes of our study we need a large number of programs that are not trivial and are accompanied with real faults. The fault set has to be large and of diverse types. Unfortunately, mutation testing is costly and its experimentation requires generating strong test suites (Titcheu Chekam et al. 2017). Therefore, there are two necessary tradeoffs, between the number of faults to be considered, the strengths of the test suites to be used and the size of the used programs.

To account for these requirements, we used the Codeflaws benchmark (Tan et al. 2017). This benchmark consists of 7,436 programs (among wich 3,902 are faulty) selected from the Codeforces[1] online database of programming contests. These contests consist of three to five problems, of varied difficulty levels. Every user submits its programs that resolve the posed problems. In total, the benchmark involves programs from 1,653 users "with diverse level of expertise" (Tan et al. 2017).

Every fault in this benchmark has two program instances: the rejected *'faulty'* submission and the accepted *'correct'* submission. Overall, the benchmark contains 3,902 faulty program versions of 40 different defect classes. It is noted that every faulty program instance in our dataset is unique, meaning that every program we use is different from the others (in terms of implementation). To the best of our knowledge, this is the largest number of

[1] http://codeforces.com/

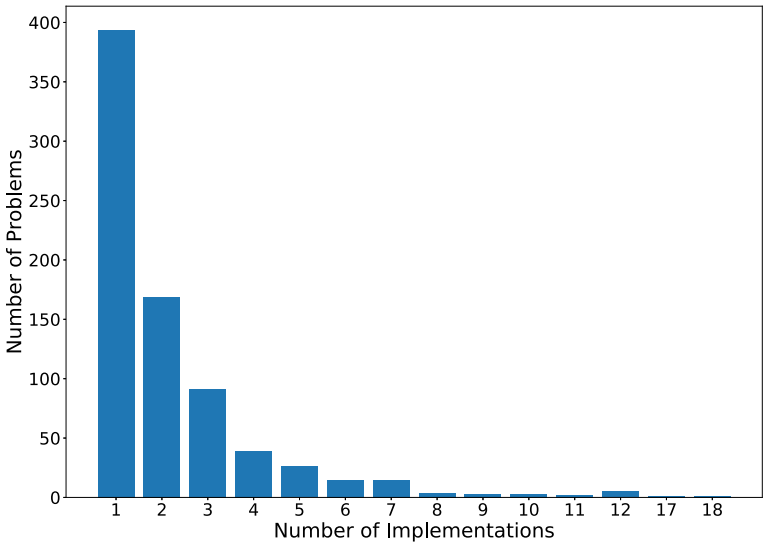


Fig. 6 Distribution of Codeflaws Benchmark problems by number of implementations

faults used in any of the mutation testing studies. The size of the programs varies from 1 to 322 with an average of 36 lines of code. Applying mutation testing on Codeflaws yielded 3,213,543 mutants and required a total of 8,009 CPU days for all computations.

To strengthen our results and demonstrate the ability of our approach to handle faults made by actual developers, we also used the CoREBench (Böhme and Roychoudhury 2014) benchmark. CoREBench includes real-world complex faults that have been systematically isolated from the history of C open source projects. These programs are of 9-83 KLoC and are accompanied by developer test suites. It is noted that every CoREBench fault forms a single fault instance (it differs from the other faults).

We used the available test suites augmented by KLEE (Cadar et al. 2008). Although these test suites greatly increased the cost of our experiment, we considered their use of vital importance as otherwise our results could be subject to "noise effects" (Titcheu Chekam et al. 2017).

Due to the very high cost of the experiments and technical difficulties to reproduce some faults, we conducted our analysis on 45 faults (22 in Coreutils, 12 in Find and 11 in Grep). Applying mutation testing on these 45 versions yielded 1,564,614 mutants and required a total of 454 CPU days of computation (without considering the test generation and machine learning computations and evaluations). Test generation resulted in a test pool composed of 122,261 and 22,477 test cases related to Codeflaws, CoREBench.

The goal of our study is to evaluate the fault revealing ability of the mutants we select. However, approximately half of our faults are trivial ones (triggered by most of the test cases), and their inclusion in our analysis would artificially inflate our results. Thus, we restrict our analysis on the faults that are revealed by less than 25% of the test cases involved in our test suites. Taking such a threshold is usual in fault injection studies (SiR 2018), but it ensures that the targeted faults and our focus is on faults that are hard enough to find. Practically, taking a lower threshold will significantly reduce the number of faults to be considered hindering our ability to train, while taking a higher threshold will make all the approaches perform similarly, as the faults will be easy to reveal. Overall, from the

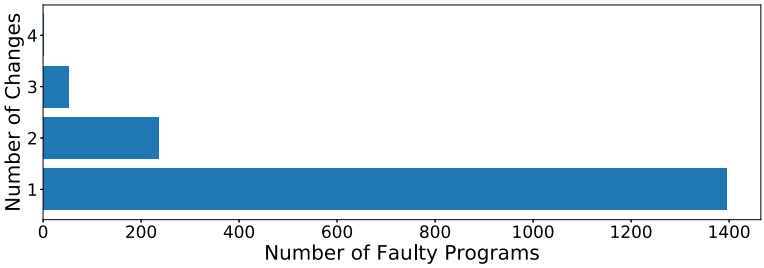


Fig. 7 Distribution of Codeflaws Benchmark faulty programs by number of lines of code changed to fix the fault

Codeflaws benchmark we consider 1,692 out of the 3,902 ones (1,692 are the non-trivial faults) and from the CoreBench benchmark 45 faults.

Figure 6 shows the distribution of number of problems by number of implementations for the considered faulty programs from Codeflaws. We observe that 85% of the problems have at most 3 implementations.

Despite that Codeflaws benchmark faults were mined from a programming contest, the faults nevertheless are relatively small syntactical mistakes. We observe on Fig. 7 that 82% of the faults are fixed by modifying a single line of source code. This ensures that we are compatible with the competent programmer hypothesis[2], which is one of the basic assumptions of mutation testing (DeMillo et al. 1978).

5.2 Automated Tools Used

We used KLEE (Cadar et al. 2008) to support the test generation. We used KLEE with a relatively large timeout limit, equal to two hours per program, the Random Path search strategy, with Randomize Fork Enabled, Max Memory 2048 MB, Symbolic Array Size 4096 elements, Symbolic Standard input size 20 Bytes and Max Instruction Time of 30 seconds. This resulted in 26,229 and 1,942 test cases for CodeFlaws and CoREBench. Since the automatically generated test cases do not include any test oracle, we used the programs' fixed version as oracle. We considered as failing, every test case that resulted in different observable program output when executed in the 'faulty' from that in the 'correct'-fixed one. Similarly, we used the program output to identify the killed mutants. We deemed a mutant as killed if it resulted in a different output than in the original program.

We built a mutation testing tool[3] that operates on LLVM bitcode. Actually all our metrics and analysis were performed on the LLVM bitcode. Our tool implements 18 operators, composed of 816 transformation rules. These include all those that are supported by modern mutation testing tools (Offutt et al. 1996a; Papadakis et al. 2018a; Coles et al. 2016) and are detailed in Table 2.

Each mutation operation consists of matching an instruction type (original instruction type) and replacing with another instruction type (mutated instruction type). Thus, a mutation operator is defined as pair of original instruction type and mutated instruction type. The

[2]The competent programmer hypothesis states that programs have a small syntactic distance from the correct version so that we need a few keystrokes to correct the program

[3]https://github.com/thierry-tct/mart

Table 2 Mutant types

Mutated instruction	Original instruction type	Mutated instruction type
STATEMENT	ANY STMT	TRAPSTMT
	ANY STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.BINARY
	SCALAR.BINARY	SCALAR.UNARY
	SCALAR.ATOM	SCALAR.BINARY
	SCALAR.BINARY	TRAPSTMT
	POINTER.BINARY	POINTER.BINARY
	SCALAR.BINARY	DELSTMT
	DEREFERENCE.BINARY	DEREFERENCE.BINARY
	SCALAR.UNARY	SCALAR.UNARY
	POINTER.BINARY	POINTER.UNARY
	DEREFERENCE.BINARY	DEREFERENCE.UNARY
	POINTER.ATOM	POINTER.UNARY
	POINTER.UNARY	POINTER.UNARY

instruction types are defined as following (p refers to pointer values and s refers to scalar values):

- **ANY STMT** refers to matching any type of statement (only original instruction type).
- **TRAPSTMT** refers to a *trap*, which cause the program to abort its execution (only mutated instruction type).
- **DELSTMT** refers to statement deletion, i.e., replacing by the empty statement which is equivalent to deleting the original statement (applies only on the mutated instruction type).
- **CALL STATEMENT** refers to a function call.
- **SWITCH STATEMENT** refers to a C language like *switch* statement.
- **SHUFFLEARGS** can only be a mutated instruction type and, when the orignal instruction type is a function call. It refers to the same function call as the original but with arguments of, same type, swapped. (e.g. $f(a, b) \rightarrow f(b, a)$)
- **SHUFFLECASESDEST** can only be used as mutated instruction type and, when the orignal instruction type is a *switch* statement. It refers to the same *switch* statement as the original but with the basic blocks of the *cases* swapped. (e.g. $\{case\ a : B_1;\ case\ b : B_2;\ default : B_3;\} \rightarrow \{case\ a : B_2;\ case\ b : B_1;\ default : B_3;\}$)
- **REMOVECASES** can only be used as mutated instruction type and, when the orignal instruction type is a *switch* statement. It refers to the same *switch* statement as the original but with some *cases* deleted (the corresponding values will lead to execute the *default* basic block). (e.g. $\{case\ a : B_1;\ case\ b : B_2;\ default : B_3;\} \rightarrow \{case\ a : B_2;\ default : B_3;\}$)

- **SCALAR.ATOM** refers to any non pointer type variable or constant (only original instruction type).
- **POINTER.ATOM** refers to any pointer type variable or constant (only original instruction type).
- **SCALAR.UNARY** refers to any non pointer unary arithmetique or logical operation (e.g. $abs(s)$, $-s$, $!s$, $s++$ ...).
- **POINTER.UNARY** refers to any pointer unary arithmetique operation (e.g. $p++$, $--p$ ...).
- **SCALAR.BINARY** refers to any non pointer binary arithmetique, relational or logical operation (e.g. $s_1 + s_2$, $s_1 \&\& s_2$, $s_1 >> s_2$, $s_1 <= s_2$ ...).
- **POINTER.BINARY** refers to any pointer binary arithmetique or relational operation (e.g. $p + s$, $p_1 > p_2$ ...).
- **DEREFERENCE.UNARY** refers to any combination of pointer dereference and scalar unary arithmetic operation, or combination of pointer unary operation and pointer dereference (e.g. $(*p)--$, $*(p--)$ ...).
- **DEREFERENCE.BINARY** refers to any combination of pointer dereference and scalar binary arithmetic operation, or combination of pointer binary operation and pointer dereference (e.g. $(*p) + s$, $*(p + s)$ ...).

Applying mutation testing on CodeFlaws and CoREBench yielded 3,213,543 and 1,564,614 mutants.

To reduce the influence of redundant and equivalent mutants, we applied TCE (Papadakis et al. 2015; Hariri et al. 2016; Kintis et al. 2018). Since we operate on LLVM bitcode we compared the mutated optimized LLVM codes using the llvm-diff utility. llvm-diff is a tool like the known Unix diff utility but for LLVM bitcode. TCE Detected 1,457,512 and 715,996 mutant equivalences on CodeFlaws and CoREBench. Note that the equivalent and redundant mutants detected by TCE are removed from the mutants set and neither executed nor considered in the experiments.

The execution of the mutants post TCE resulted in killing the 87% and 54% of the mutants for CodeFlaws and CoREBench. It is important to note that our tool applies mutant test execution optimizations by recording the coverage and program state at the mutation points avoiding the execution of mutants that do not infect the program state (Papadakis and Malevris 2010a). This optimization enables huge test execution reductions and forms the current state of the art at the test execution optimizations (Papadakis et al. 2018a). Despite these optimization our tool required a total of 8,009 and 454 CPU days of computations for CodeFlaws and CoREBench indicating the large amount of computation resources required to perform such an experiment.

5.3 Experimental Procedure

To answer our research questions we performed an experiment composed of three parts. The first part regards the prediction ability of our classification method, answer RQ1 and RQ2, the second regards the fault revealing ability of the approaches, answer RQ3-RQ6, and the third regards the fault revealing ability of our approach on large independently selected programs, answer RQ7. To account for our use case scenario, in our experiments we always train and evaluate our approach on a different sets of programs (CodeFlaws) or program versions (CoREBench).

As a first step we used KLEE to generate test cases for all the programs we study and formed a pool of test cases by joining the generated and the available test cases. We then

constructed a mutation-fault matrix, which records for every test case the mutants that it kills and whether it reveals the fault or not (we construct a matrix for every single fault we study). We also record the execution time needed to execute every mutant-test pair so that we can simulate the execution cost of the approaches. We make the data available[4].

To measure fault revelation we mutated the faulty program versions. This is important in order to avoid making any assumption related to the interaction of mutants and faults, aka Clean Program Assumption (Titcheu Chekam et al. 2017). Based on this matrix we compute the fault revealing ratio for each mutant. The *fault revealing ratio* is the ratio of tests that kill the mutant and reveal the fault to the total number of tests that kill the mutant.

First experimental part: The first task of prediction modeling is to evaluate the contribution of the used features. We computed the information gain values for each one of the used features. Higher information gain values represent more informative features for decision trees. Demonstrating the importance of our features helps us understand what is the most important factors affecting the utility of mutants. Having measured information gain, we then measure the prediction ability of our classification method by evaluating its ability to predict killable and fault revealing mutants. For this part of the experiment we considered as fault revealing the mutants that have fault revealing ratio equal to 1. We relax this constraint in the second part of the experiment.

We evaluate the trained classifiers using four typically adopted metrics such as the precision, recall, F-measure and Area Under Curve (AUC). The *precision* of a classifier is defined as the number of items that are truly relevant among the items that the classifier predicted to be relevant. The *recall* of a classifier is defined as the number of items that are predicted to be relevant by the classifier among all the truly relevant items. The F-measure of a classifier is defined as the weighted harmonic mean of the precision and recall, it is also named *F1 score*. The Area Under Curve (AUC) of a classifier is the area under the Receiver Operating Characteristic (ROC) curve (The ROC curve shows how many true positive classifications can be gained as more and more false positives are allowed) (Zheng 2015). Precision represents the ratio of the identified killable and fault revealing mutants out of those classified as such. Recall represents the ratio of the identified killable and fault revealing mutants out of all existing ones. In classification usually recall and precision are competitive metrics in the sense that higher values of one imply lower values for the other. To better compare classifiers researchers use the F-measure and AUC metrics. These measure the general classification accuracy of the classifier. Higher values denote better classification.

To reduce the risk of overfitting, we applied a 10-fold cross validation by partitioning our program set into 10 parts and iteratively train on 9 parts and evaluation on one. We report the results for all the partitions.

This experiment part was performed on the Codeflaws programs.

Second experimental part: Our analysis requires comparing mutation-based strategies with respect to the actual value of interest, the number of faults revealed. Given that killing a mutant does not always result in revealing a fault, we train the classifier in accordance with the actual fault revealing ratios (i.e., the ratio of tests that kill a mutant and also reveal faults).

We then select and prioritise our mutants. To evaluate and compare the studied approaches with respect to fault revelation, we follow a typical procedures (Titcheu Chekam et al. 2017; Kurtz et al. 2016; Namin et al. 2008) by randomly selecting test cases, from the formed test pools, that kill the selected mutants. In case none of the available test cases on

[4]https://mutationtesting.uni.lu/farm

our test pool kills the mutant we treat it as equivalent. We repeat this process for each one of the studied approaches. As done in the first part of the experiment we report results using a 10-fold cross validation.

For the mutant selection problem we randomly pick a mutant and then randomly pick a test case that kills it. Then we remove all the killed mutants and pick another one. If the mutant is not killed by any of the test cases on our test pool we treat it as equivalent. We repeat this process 100 times and compute the probability of revealing each one of the faults.

For the mutant prioritisation case we follow the mutant order by picking test cases that kills each mutant. We do not attempt to kill a mutant twice. Again, we repeat this process 100 times and compute the Average Percentage of Faults Detected (APFD) values, which is typical metric used test case prioritization studies (Henard et al. 2016). Again we align the compared approaches with respect to their cost (number of mutants need manual analysis) and compare their effectiveness.

To account for coincidental results and the stochastic selection of test cases and mutants we used the Wilcoxon test, which is a non-parametric test, to determine whether the Null Hypothesis (that there is no difference between the studied methods) can be rejected. In case the Null Hypothesis is rejected, then we have evidence that our approach outperforms the others. Even when the null hypothesis does not hold, the size of the differences might be small. To account for this effect we also measured the Vargha Delaney effect size $\hat{A}_{12}$ (Vargha and Delaney 2000), which quantifies the size of the differences (aka statistical effect size). $\hat{A}_{12} = 0.5$ suggests that the data of the two samples tend to be the same. $\hat{A}_{12} > 0.5$ values indicate that the first dataset has higher values, while $\hat{A}_{12} < 0.5$ indicate the opposite.

This experiment part was performed on the Codeflaws programs.

Third experimental part: To further evaluate the fault revealing ability of our approach, we applied it on the CoreBench programs. We also adopted the 10-fold cross validation as for the experiments on Codeflaws. We report results related to both fault revelation and APFD values. The CoreBench corpus is small in size and hence *FaRM* might not be particularly important. However, in case the signal of our features is strong, we will be able to experience the benefits of our method even with those few data.

5.4 Mutant Selection and Effort Metrics

When comparing methods, a comparison basis is required. In our case we measure fault revelation and effort. While measuring fault revelation based on the fault set we use is direct, measuring effort/cost is hard. Effort/cost depends on a large number of uncontrolled parameters, such as the followed procedure, level of automation, skills, underlying infrastructure and the learning curve. Therefore, we have to account for different scenarios. and we adopt three frequently used metrics; the number of selected mutants, the number of test cases generated and the number of mutants requiring analysis.

The first metric (selected mutants) represents the number of mutants that one should use when applying mutation testing. This is a direct and intuitive metric as it suggest that developers should select a particular set of mutants to generate (form an actual executable codes), execute and analyse. Although such a metric conforms to our working scenario, it does not focus on the required test generation effort involved. Generating test cases is mostly a manual task (due to the test oracle problem) and so, we also consider a second metric, the number of test cases that can be generated based on a selected set of mutants.

We also adopt a third metric, the number of mutants that need to be analysed (equivalent mutants and those we pick, i.e., analysed in order to generate test cases). This metric somehow reflects the effort a tester needs to put in order to kill or identify as equivalent the

selected mutants (under the assumption that equivalent mutants require the same effort as the test generation).

To fairly compare the random selection methods, we select mutants until we analyse the same number of mutants as analysed by our selection method. This establishes a fixed cost point for all the approaches and compare their effectiveness.

There are other cost factors, such as the mutant-test execution cost and the analysis of equivalent mutants (for the first two metrics) that we investigate separately. The reason for that is that we would like to see if our approaches are also faster to execute and require reasonably less equivalent mutants.

6 Results

6.1 Assessment of Killable Mutant Prediction (RQ1 and RQ2)

To check the prediction performance of our classifier we performed a 10-Fold cross-validation for three different selected sets. These were the results of applying *PredKillable* to predict killable mutants and selecting the 5%, 10% and 20% of the top ranked mutants. The *PredKillable* classifier achieves 98.8% 5.7%, 10.7% precision, recall and F-measure when selecting the 5% of the mutants. With respect to 10% and 20% sets of mutants, it achieves 98.8% and 98.7% (precision), 11.4% and 22.8% (recall), 20.4% and 37.0% F-measures. These values are higher than those that one can get by randomly sampling the same number of mutants. In particular the *PredKillable* has 12.3%, 12.2% and 12.1% higher precision, and 0.7%, 1.4% and 2.8% higher recall values for the 5%, 10% and 20% sets of mutants.

When using *PredKillable* to predict non killable mutant, the classifier achieves 95.1% 35.0%, 51.2% precision, recall and F-measure when selects the 5% of the mutants. With respect to 10% and 20% sets of mutants, it achieves 79.1% and 49.3% (precision), 58.6% and 73.2% (recall), 67.3% and 58.9% F-measures. These values are higher than those that one can get by randomly sampling the same number of mutants. In particular the *PredKillable* has 81.6%, 65.7% and 35.8% higher precision, and 30.1%, 48.7% and 53.3% higher recall values for the 5%, 10% and 20% sets of mutants.

To train our models, approximately 48 CPU hours were required, while to perform the evaluation (perform mutant selection) it required less than a second. Since training should only happen once in a while, the training time is considered acceptable. Of course the cost of selecting and prioritizing mutants is practically negligible.

The Receiver operating characteristic (ROC) shown in Fig. 8 further illustrates performance variations of the classifier in terms of true positive and false positive rates when the discrimination threshold changes: the higher the area under curve (AUC), the better the classifier. Our classifier achieves an AUC of 88%. These results establish that the code properties that were leveraged as features for characterizing mutants provide, together, a good discriminative power for assessing the fault revealing potential of mutants.

6.2 Assessment of Fault Revelation Prediction

ML prediction performance Similarly to Section 6.1 we performed a 10-Fold cross-validation for three different selected sets in order to check the prediction performance of our classifier. These were the results of applying *FaRM* and selecting the 5%, 10% and 20% of the top ranked mutants. The *FaRM* classifier achieves 5.7% 12.8%, 7.8% precision, recall

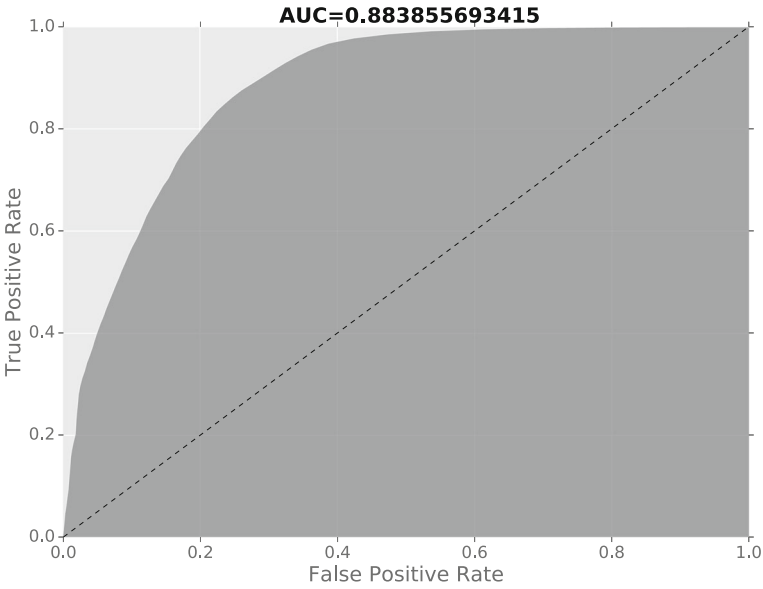


Fig. 8 Receiver Operating Characteristic For Killable Mutants Prediction on Codeflaws

and F-measure when selects the 5% of the mutants. With respect to 10% and 20% sets of mutants, it achieves 4.9% and 3.9% (precision), 22.0% and 35.1% (recall), 8.0% and 7.0% F-measures. These values are higher than those that one can get by randomly sampling the same number of mutants. In particular *FaRM* has 3.5%, 2.7% and 1.7% higher precision, and 7.8%, 12.1% and 15.1% higher recall values for the 5%, 10% and 20% sets of mutants.

The cost of training and evaluation are same as those reported in Section 6.1.

The Receiver operating characteristic (ROC) shown in Fig. 9 further illustrates performance variations of the classifier in terms of true positive and false positive rates when the discrimination threshold changes: the higher the area under curve (AUC), the better the classifier. Our classifier achieves an AUC of 62%.

We believe that such a result is encouraging due to the nature of the developer mistakes. As developers make mistakes in an non-systematic way, for the same problem, some may make mistakes while some others may not, the only thing we can hope for is to form good heuristics, i.e., identify mutants that maximsize the chances to reveal faults. Therefore, it is hard to get much higher AUC values. Nevertheless, we expect future research to built on and improve our results by forming better predictors.

Overall, the above results demonstrate that the code properties that were leveraged as features for characterizing mutants provide, together, a discriminative power to assess the fault revealing potential of mutants.

Considered features We provide in Fig. 10 the distribution of information Gain values for the various features considered in this work. Information gain (IG) measures how much "information" a feature gives us about the class we want to predict. The IG values are computed by the supervised learning algorithm during the training process. These data enable the assessment of the potential contribution of every feature to a prediction model. Experimental training process provides evidence in Fig. 10 that the suggested features (in bold) contribute significantly less than several other features that we have designed for *FaRM* .

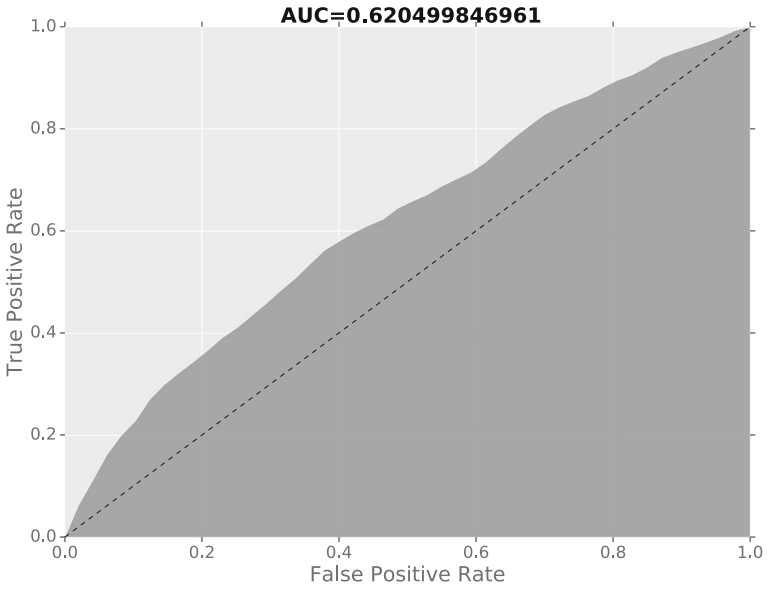


Fig. 9 Receiver Operating Characteristic For Fault Revealing Mutants Prediction on Codeflaws

Interestingly, together with complexity, the features related to control and data dependencies are the most informative ones. Here we should note that IG values do not suggest which features to select and which not. Actually our results show that we need all the features.

6.3 Mutant Selection

6.3.1 Comparison with Random (RQ3)

Figure 11 shows the distribution of the fault revelation of the mutant selection strategies when selecting the 2%, 5% and 10% of the top ranked mutants. As can be seen from the plot, both *FaRM** and *FaRM* outperforms both DummyRandom and spreadRandom. Both DummyRandom and spreadRandom outperform *PredKillable* . When selecting 2% of the mutants the difference, for both *FaRM* and *FaRM** , of the median values is 22% and 24%

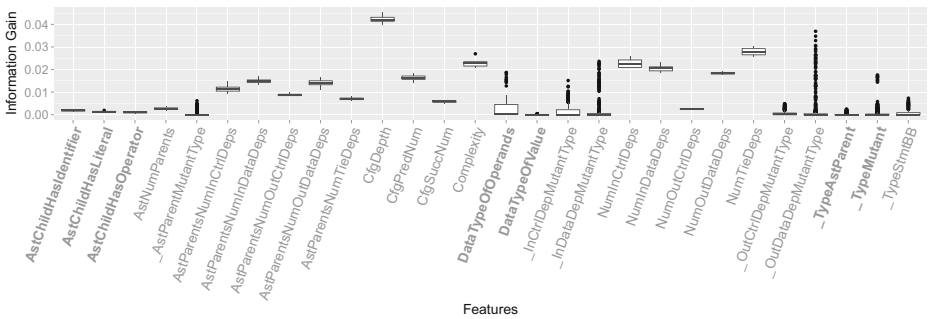


Fig. 10 Information Gain distributions of ML features on Codeflaws

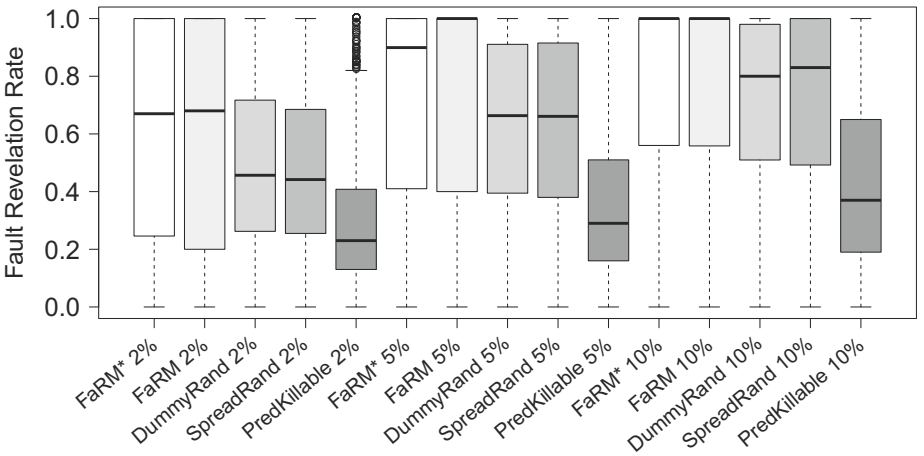


Fig. 11 Fault revelation of the mutant selection strategies on Codeflaws. All three *FaRM* and *FaRM** sets outperform the random baselines

for the DummyRandom and SpreadRandom respectively. This difference is increasing when selecting the 5% of the mutants and goes to 34% and 34% for *FaRM* and, 24% and 24% for *FaRM** . When selecting 10% of the mutants the difference becomes 20% and 17% for both *FaRM* and *FaRM** . Regarding *PredKillable* , the difference with DummyRandom and SpreadRandom at the 2% mutant selection threshold is 23% and 21% respectively. This difference increase for the 5% to 37% and 37%. For the 10% threshold is 43% and 46%.

To check whether the differences are statistically significant we performed a Wilcoxon rank-sum test, which is a non-parametric test that measures whether the values of one sample are higher than those of the second sample. We adopt a statistically significant level $a < 0.01$ below of which we consider the differences as statistically significant. We also computed the Vargha Delaney $\hat{A}_{12}$ effect size value between the approaches.

The statistical test showed that *FaRM* and *FaRM** outperforms both DummyRandom and SpreadRandom with statistically significant difference. both DummyRandom and SpreadRandom outperform *PredKillable* with statistically significant difference. As expected the differences between DummyRandom and SpreadRandom are not significant. It is noted that all comparisons are aligned with respect to the number of mutants that need analysis, which as we already explained represents the manual effort involved. The Vargha Delaney $\hat{A}_{12}$ value between the approaches show that for the 2% threshold, *FaRM* is better than DummyRandom and SpreadRandom in 60% and 63% of the cases respectively. These values are slightly higher for *FaRM** where it is better than DummyRandom and SpreadRandom in 62% and 65% of the cases respectively. DummyRandom and SpreadRandom are respectively better than *PredKillable* in 84% and 82% of the cases. For the 5% threshold, *FaRM* is better than DummyRandom and SpreadRandom in 66% of the cases. *FaRM** is better than DummyRandom and SpreadRandom in 64% and 65% of the cases respectively. DummyRandom and SpreadRandom are respectively better than *PredKillable* in 88% and 84% of the cases. For the 10% threshold, *FaRM* is better than DummyRandom and SpreadRandom in 65% and 63% of the cases respectively. *FaRM** is better than DummyRandom and SpreadRandom in 64% and 61% of the cases respectively. DummyRandom and SpreadRandom are respectively better than *PredKillable* in 87% and 85% of the cases.

Regarding the test execution time of the involved methods, our approach has an advantage but this is minor. The median difference between *FaRM* and DummyRandom and SpreadRandom was 12 and 39 seconds per program respectively. This means that *FaRM* required 12 and 29 seconds less execution time than the random baselines. While these differences are considered as minor they demonstrate that *FaRM* has significantly higher fault revelation ability than the compared baselines without introducing any major overhead.

Overall, our results suggest that *FaRM* and *FaRM** significantly outperforms the random baselines with practically significant differences, i.e., improvements on the ratios of revealed faults were between 4% to 34%. *PredKillable* is outperformed by all the approaches.

6.3.2 Comparison with SDL & E-selective (RQ4)

This section aims to compare the fault revelation of our approach with that of the SDL and the E-Selective mutants sets.

In order to compare our approach with SDL selection, the selection size is set to the number of SDL mutants. In the Codeflaws subjects, SDL and E-SELECTIVE mutants represent in median respectively 2% and 38% of all mutant as seen in Fig. 12.

Our analysis is designed as following. For each subject, the $|SDL|$ top ranked mutants of *FaRM* are selected (where $|SDL|$ is the total number of SDL mutants). We also select the $|SDL|$ top ranked mutants with the random approaches. Then, the fault revelation of each approach's selected mutants set is computed for comparison and presented in Fig. 13. We observe that *FaRM* and *FaRM** respectively have 30% and 27% higher median fault revelation than SDL. *PredKillable* has 25% lower median fault revelation than SDL. We also observe that SDL has similar fault revelation with the random selections (respectively 3% and 2% lower than DummyRandom and SpreadRandom).

We also performed the Wilcoxon rank-sum test as in Section 6.3. The statistical test showed that both *FaRM* and *FaRM** outperform SDL, and SDL outperforms *PredKillable* . The difference between SDL and DummyRandom and SpreadRandom is not statistical significant. We also computed the Vargha Delaney $\hat{A}_{12}$ value between the approaches and found that *FaRM* and *FaRM** are respectively better than SDL in 54% and 55% of the cases. SDL is better than *PredKillable* in 79% of the cases.

Similar to the experiment performed above to compare our approach with SDL, we perform another experiment to compare *FaRM* with E-Selective selection. The fault revelation

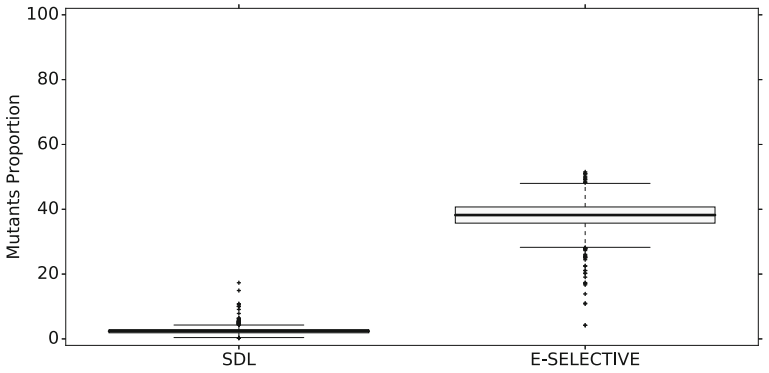


Fig. 12 Proportion of SDL and E-SELECTIVE mutants among all mutants for Codeflaws subjects

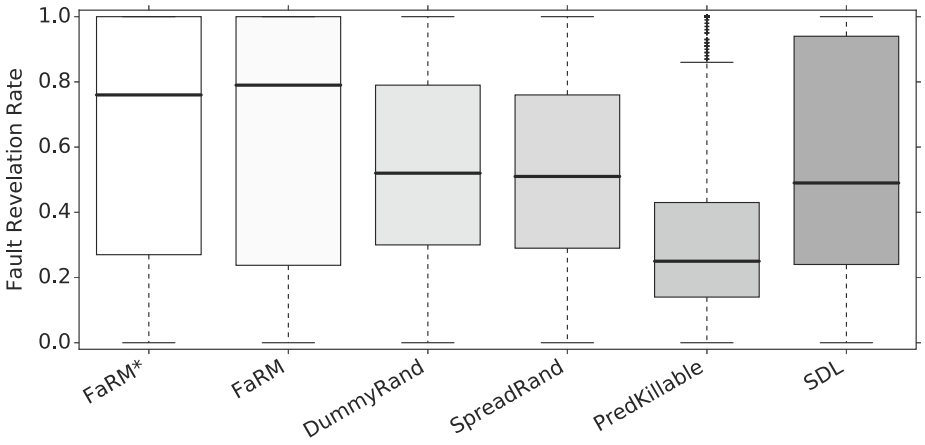


Fig. 13 Fault revelation of *FaRM* compared with SDL on Codeflaws. *FaRM* sets outperform the SDL selection. Approximately 2% (number of SDL mutants) of all the mutants are selected

results are presented in Fig. 14. We observe that for a selection size equal to the number of E-Selective mutants, all selection approaches except *PredKillable* and DummyRandom achieve the highest median fault revelation. Given that E-Selective mutants are roughly 38% of all the mutants, which is relative large set, we make the comparison with the E-Selective set for smaller selection size namely 5% and 15% thresholds of the top ranked mutants (w.r.t all mutants). The E-Selective mutants of the given sizes are randomly selected from the whole E-Selective mutant set. The fault revelation results are presented in Figs. 15 and 16. We can observe that *FaRM* and *FaRM** respectively have 31% and 22% higher median fault revelation than E-Selective for thresholds 5%. For the 15% threshold, both have 9% higher median fault revelation. *PredKillable* has 38% and 47% lower median fault revelation than E-Selective for thresholds 5% and 15% respectively. We also observe that E-Selective has similar fault revelation with the random selections (respectively 2% and 1% higher than

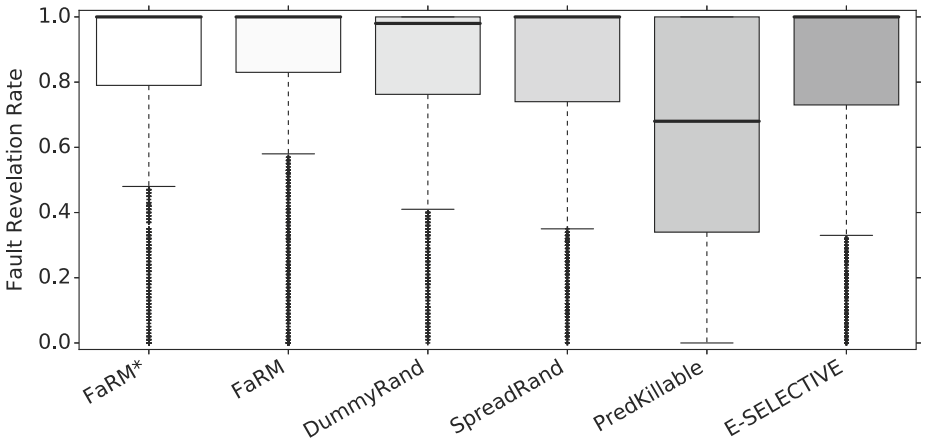


Fig. 14 Fault revelation of *FaRM* compared with E-Selective on Codeflaws. Approximatively 38% (number of E-Selective mutants) of all the mutants are selected

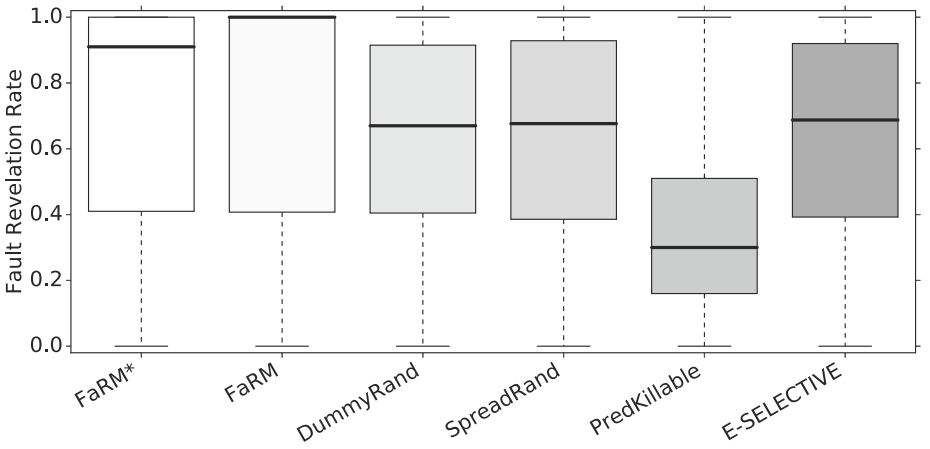


Fig. 15 Fault revelation of *FaRM* compared with E-Selective for selection size 5% of all mutants. *FaRM* and *FaRM** sets outperform E-Selective selection

DummyRandom and SpreadRandom for selection size 5% and respectively 3% and 0% higher than DummyRandom and SpreadRandom for selection size 15%).

The Wilcoxon rank-sum statistical test shows that both *FaRM* and *FaRM** outperform E-Selective, and E-Selective outperforms the *PredKillable* . The difference between E-Selective and the random approaches is not statistical significant. We also computed the Vargha Delaney $\hat{A}_{12}$ effect size value between the approaches and found that for the 5% and 15% thresholds, *FaRM* is better than E-Selective in 64% and 63% of the cases respectively. *FaRM** is better in 62% and 61% of the cases respectively, and *PredKillable* is worse in 86% and 82% of the cases respectively.

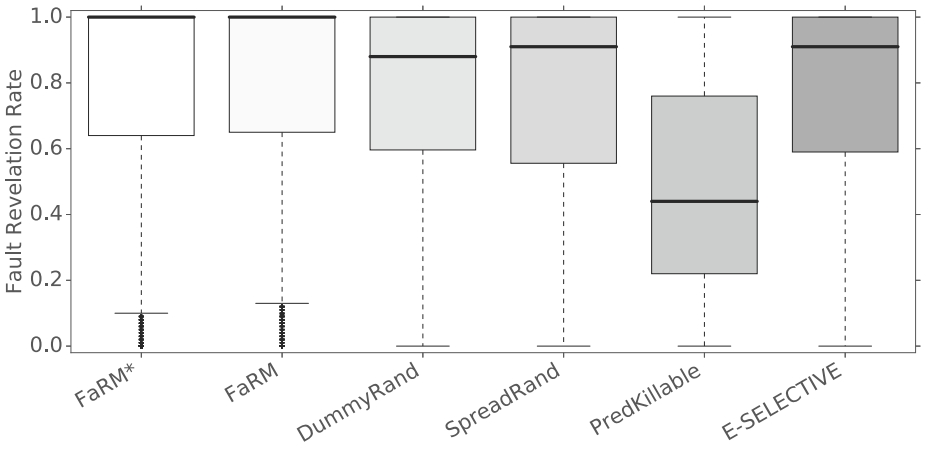


Fig. 16 Fault revelation of *FaRM* compared with E-Selective for selection size 15% of all mutants. *FaRM* sets outperform E-Selective selection

6.4 Mutant Prioritization

6.4.1 Comparison with Random (RQ5)

Selected Mutants Cost Metric Figure 17 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the five approaches under evaluation. While *FaRM* and *FaRM** respectively yield an APFD median of 98% and 97%, and *PredKillable* yields an APFD median of 72%, DummyRandom and SpreadRandom reach median APFD values of 93% and 94% respectively. These results reveal that the general trend is in favour to our approach. As our approaches *FaRM* and *FaRM** are better than the random baseline, when the main cost factor (number of mutants that need analysis) is aligned, we can infer that it is generally better with practically important differences (of 4%). Note that the highest possible improvement over the random baseline is 6% (DummyRandom has a median APFD value of 94%). Nonetheless, *PredKillable* is worse than the random baseline.

To account for the stochastic nature of the compared methods and increase the confidence on our results, we further perform a statistical test. Wilcoxon test results yielded p-values much lower than our significance level for the compared data, i.e., samples of *FaRM* and DummyRandom, *FaRM* and SpreadRandom, *FaRM** and DummyRandom, *FaRM** and SpreadRandom, *PredKillable* and DummyRandom, and *PredKillable* and SpreadRandom respectively. Therefore, we can definitively conclude that *FaRM* and *FaRM** outperform random mutant selection with statistically significance while random mutant selection outperform *PredKillable* . On the other hand, as expected, the Wilcoxon test revealed that there is no statistical difference between the performance of DummyRandom and that of SpreadRandom.

When examining mutant selection strategies there are two main parameters that influence the application cost. These are the killable and equivalent mutants that testers need to analyse. When analysing a killable mutant our ability to select fault revealing ones is important, while increasing the chance to get a killable mutant is also important. Therefore, it could be that our *FaRM* is better simply because it selects killable mutants and not fault revealing ones. To account for this factor we removed all non-killable mutants from our sets and

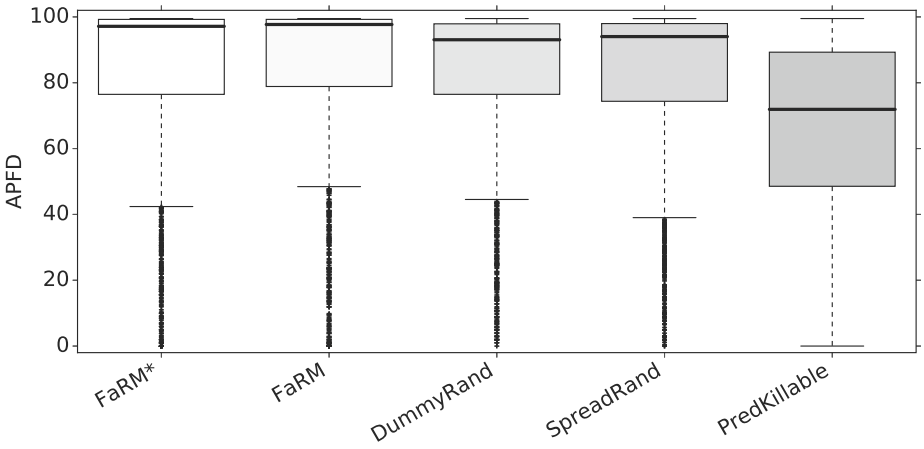


Fig. 17 APFD measurements considering all mutants for the selected mutants cost metric for Codeflaws. The *FaRM* prioritization outperform the random baselines

recompute our results. This helped eliminating the influence of non-killable mutants, from both approaches.

Our results show that the performance improvement of *FaRM* and *FaRM** over SpreadRandom and DummyRandom is also effective when considering only killable mutants (approximated by our test suites). Figure 18 shows the relevant distributions of APFD, which are visibly similar to the distributions for all mutants (all values are slightly higher when considering only killable mutants). This result suggest that *FaRM* and *FaRM ** are indeed capable of identifying fault revealing mutants, independent of the equivalent mutants involved.

To provide a general view of the trends, Fig. 19 illustrates the overall (median) effectiveness of the mutant prioritization by *FaRM* , *FaRM** and *PredKillable* in comparison with random strategies. We note that for all percentages of mutants, *FaRM* and *FaRM** outperforms random-based prioritization while *PredKillable* is outperformed by the random-based prioritization. Overall, we observe that the fault revelation benefit of *FaRM* over the random approaches is above 20% (maximum difference is 34%) when selecting 2% to 8% of mutants. *FaRM* reaches a plateau around 5% of mutants, as the median fault revelation is maximal. This suggests that a hint for the mutant selection size for *FaRM* could be 5% of the mutants.

Finally, we examined the differences between the approaches in terms of execution time. Although we do not explicitly aim at reducing the test execution cost, we expect some benefits due to our methods' ability to prioritise the mutants, which results in a reduced execution time (Zhang et al. 2013). Figure 20 illustrate, in a box-plot form, the overall execution time differences between the *FaRM* and the random baselines with respect to the attained fault revelation, measured in seconds. Although the differences can be significant in some (rare) cases, the expected (median values) ones are -58,167 and -29,373 seconds (-16 and -8 hours) for DummyRandom and SpreadRandom. This result indicates that our approach has also an advantage with respect to test execution, which sometimes becomes significant.

Conclusively, our results demonstrate that *FaRM* is indeed effective as it is statistically superior to random baselines, independent of the equivalent mutants involved. It provides

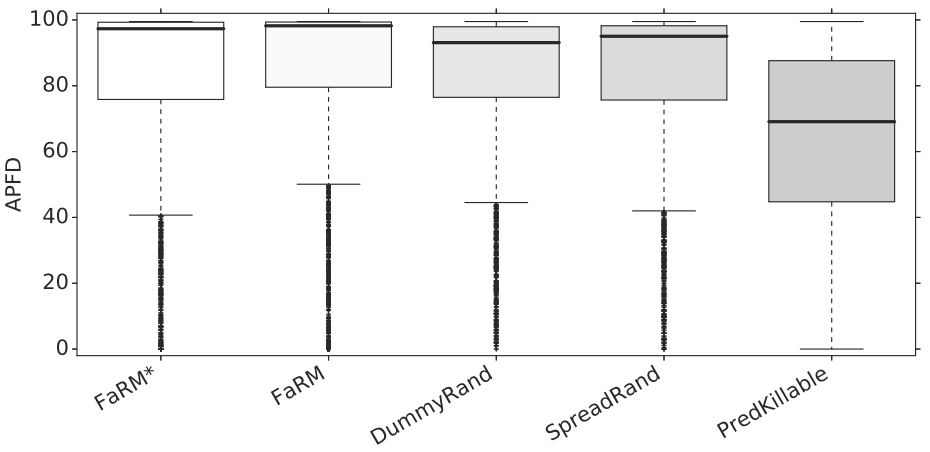


Fig. 18 APFD measurements considering only killable mutants for the selected mutants cost metric on Codeflaws. The *FaRM* prioritization outperform the random baselines, independent of non-killable mutants

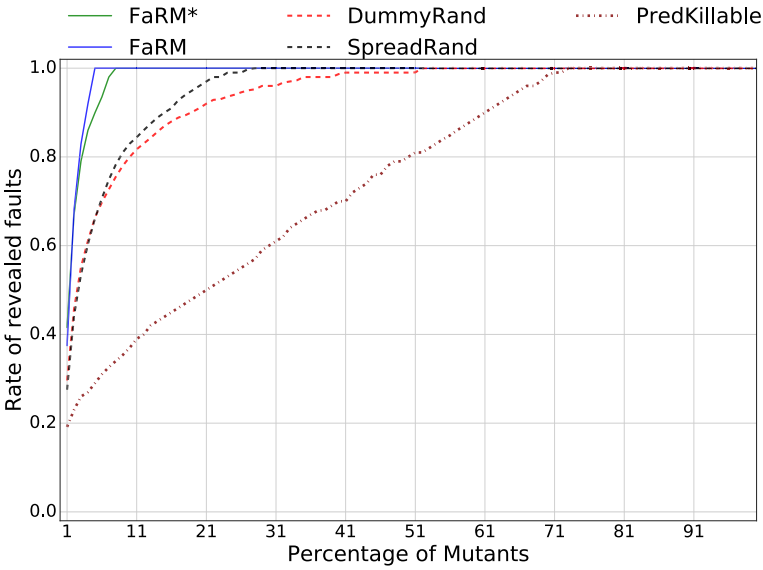


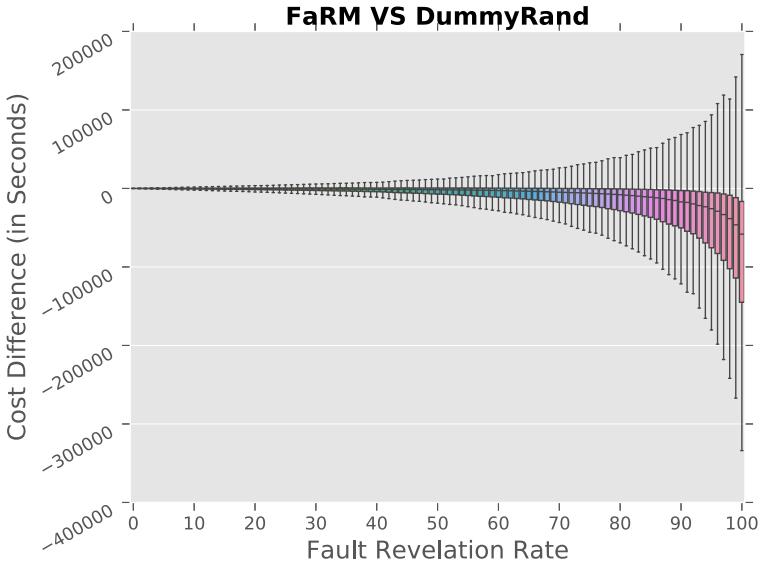
Fig. 19 Mutant prioritization performance in terms of faults revealed (median case) for the selected mutants cost metric on CodeFlaws. The x-axis represent the number of considered mutants. The y-axis represent the ratio of the fault revealed by the strategies

4% higher APFD values, which means that when testers analyse mutants (to strengthen their test suites) they get a 4% improvement on their fault revelation ability. Note that the highest possible improvement over the random baseline is 6% (DummyRandom has a median APFD value of 94%).

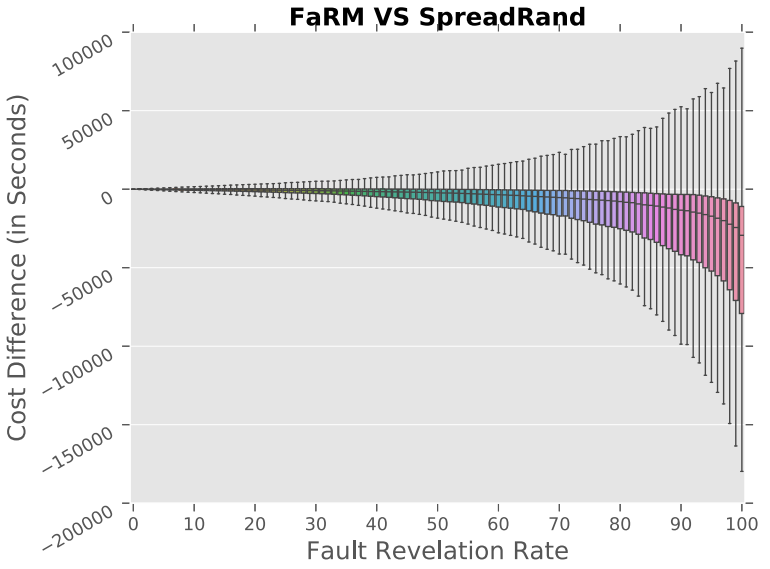
Required Tests Cost Metric Figure 21 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the five approaches under evaluation. While both *FaRM* and *FaRM** yield an APFD median of 81%, and *PredKillable* yields an APFD median of 76%, DummyRandom and SpreadRandom reach median APFD values of 77%. These results reveal that the general trend is in favour to our approach. As our approaches *FaRM* and *FaRM** are better than the random baseline, when the main cost factor (number of test that need to be designed and executed) is aligned, we can infer that it is generally better with practically important differences (of 4%). The *PredKillable* performs quite similarly to the random baseline.

To account for the stochastic nature of the compared methods and increase the confidence on our results, we further perform a statistical test. Wilcoxon test results yielded p-values much lower than our significance level for the compared data, i.e., each of *FaRM* and *FaRM** compared with each of *PredKillable* , DummyRandom and SpreadRandom. Therefore, we can definitively conclude that *FaRM* and *FaRM** outperform random baseline with statistically significance. On the other hand, the Wilcoxon test revealed that there is no statistical difference between the performance of *PredKillable* , DummyRandom and SpreadRandom.

The results of the Vargha Delaney effect size show that *FaRM* is better than DummyRandom, SpreadRandom and *PredKillable* in 58%, 61% and 60% of the cases respectively. *FaRM** is better than DummyRandom, SpreadRandom and *PredKillable* in 58%, 61% and 59% of the cases respectively.



(a) Cost of *FaRM*- Cost of DummyRand.



(b) Cost of *FaRM*- Cost of SpreadRand

Fig. 20 Execution cost of prioritization schemes

To provide a general view of the trends, Fig. 22 illustrates the overall (median) effectiveness of the required test prioritization by *FaRM* , *FaRM** and *PredKillable* in comparison with random strategies. We note that for all percentages of tests, *FaRM* and *FaRM** outperforms random-based prioritization while *PredKillable* is outperformed by the random-based

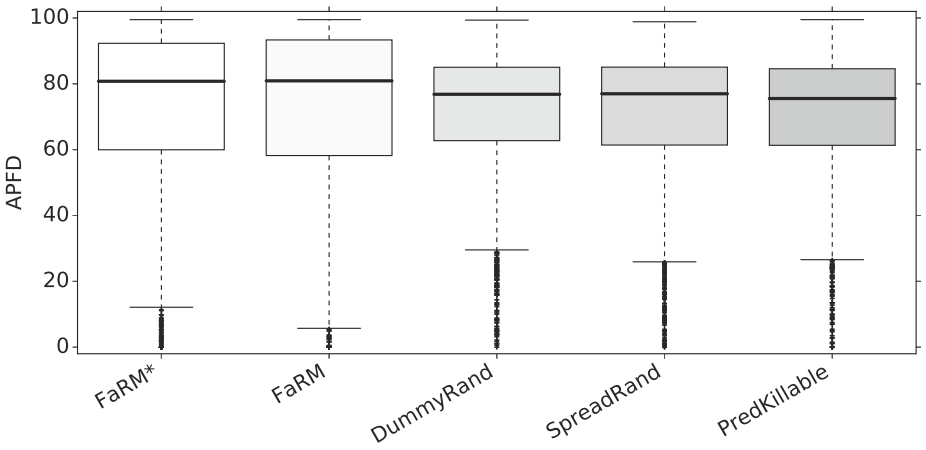


Fig. 21 APFD measurements for the required tests cost metric on Codeflaws. The *FaRM* prioritization outperform the random baselines

prioritization. Overall, we observe that the fault revelation benefit of *FaRM* over the random approaches is above 10% (maximum difference is 15%) for the 20% to 45% top ranked tests.

Analysed Mutants Cost Metric The analysed mutants cost metric measures the minimum number of mutants that need to be analysed, including equivalent mutants, following a mutant prioritization approach, before the fault is revealed. A good mutant prioritization

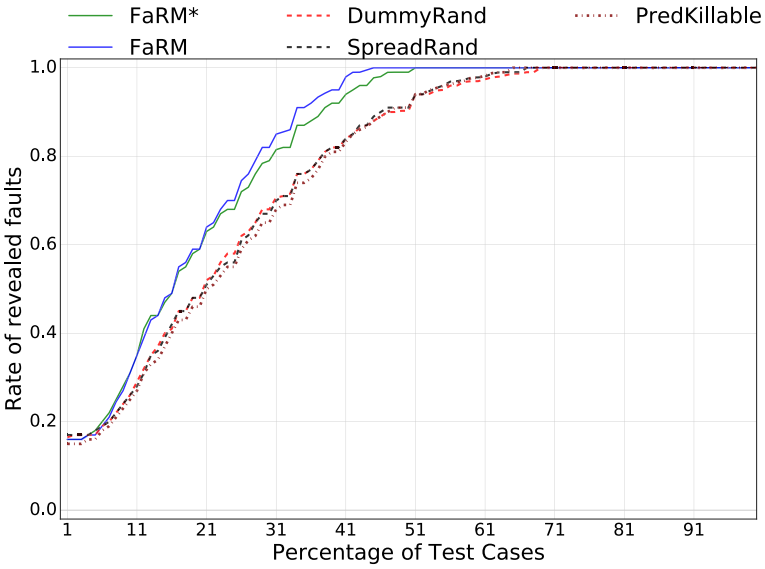


Fig. 22 Required tests prioritization performance in terms of faults revealed (median case) on CodeFlaws. The x-axis represent the number of considered tests. The y-axis represent the ratio of the fault revealed by the strategies

approach will minimized the analysed mutants cost. Following, we compare the analysed mutants cost metric between our approaches and the random baselines. The analysed mutants cost metric is calculated for each approach and for each bug of the benchmark. We compare the approaches statistically with Wilcoxon rank-sum test and the Vargha Delaney effect size.

The results show that *FaRM* , *FaRM** and *PredKillable* are better than DummyRandom and SpreadRandom with statistical significance displayed by a p-value much lower than the significance level. *FaRM* is better than DummyRandom and SpreadRandom in 57% and 61% of the cases respectively. The performance difference is higher for *FaRM** where it is better than DummyRandom and SpreadRandom in 60% and 64% of the cases respectively. *PredKillable* is better than DummyRandom and SpreadRandom in 60% and 65% of the cases respectively.

*FaRM** shows a larger improvement than *FaRM* over the random baseline, but there is no statistical significance difference between *FaRM* and *FaRM** . Furthermore, *FaRM** outperforms *PredKillable* with statistical significant difference, and is better in 53% of the cases. There is no statistical significant difference between *FaRM* and *PredKillable* .

Conclusively, our results demonstrate that *FaRM* and *FaRM** are indeed effective as they are statistically superior to random baselines.

6.4.2 Comparison with Defect Prediction (RQ6)

Selected Mutants Cost Metric Figure 23 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the *FaRM* , *FaRM** , *PredKillable* and the random approaches. While *FaRM* yields an APFD median of 98.0%, defect prediction (DefectPred) reach median APFD value of 83.7%. These results reveal that the general trend is in favour to our approach. As our approach is much better than the defect prediction approach, when the main cost factor (number of mutants that need analysis) is aligned, we can infer that it is generally better with practically important differences (of 14%). Even the random approaches are better than the defect prediction approach. Nevertheless, *PredKillable* is worse than defect prediction.

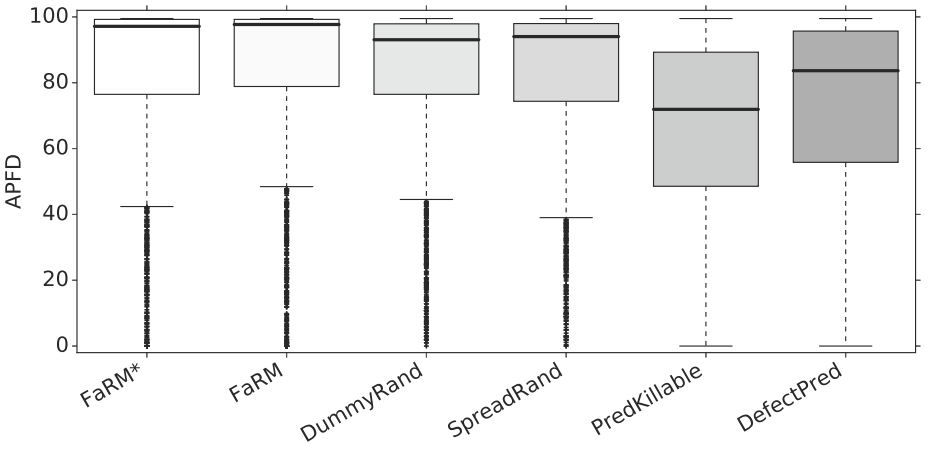


Fig. 23 APFD measurements considering all mutants. The *FaRM* prioritization outperform the defect prediction

The Wilcoxon test results yielded p-values much lower than our significance level for the samples of *FaRM* and DefectPred, and *FaRM** and DefectPred. Therefore, we can definitively conclude that *FaRM* and *FaRM** outperforms defect prediction with statistical significance. On the other hand, the Wilcoxon test also revealed that there is statistical significant difference between the performance of DefectPred and dummyRandom, and DefectPred and that of spreadRandom respectively. Nonetheless, DefectPred outperforms *PredKillable* with statistical significance. The Vargha Delaney $\hat{A}_{12}$ effect size value shows that *FaRM* and *FaRM** are better than DefectPred in 76% of cases. While DummyRandom and SpreadRandom are better than DefectPred in 71% and 70% of the cases respectively.

To provide a general view of the trends, Fig. 24 illustrates the overall (median) effectiveness of the mutant prioritization by *FaRM* in comparison with the defect prediction approach. We note that for all percentage of mutants, *FaRM* outperforms the defect prediction approach. The performance improvement goes around 40% to 66% of more faults revealed when 2% until 8% of mutants are executed.

6.5 Experiments with Large Programs (RQ7)

Selected Mutants Cost Metric in CoREBench, all APFDs values are much higher than in CodeFlaws, with *FaRM* , *FaRM** , DummyRandom and SpreadRandom having median APFD value of 99%, and *PredKillable* a median APFD value of 94%. The maximum possible improvement is 1% (given that the random baseline has a median of 99%). This is caused by the large number of redundant mutants involved. To demonstrate this we check the relation between mutation score and percentage of considered mutants. Figure 27 illustrates the overall (median) mutation score achieved (y-axis) by the tests killing the percentage of

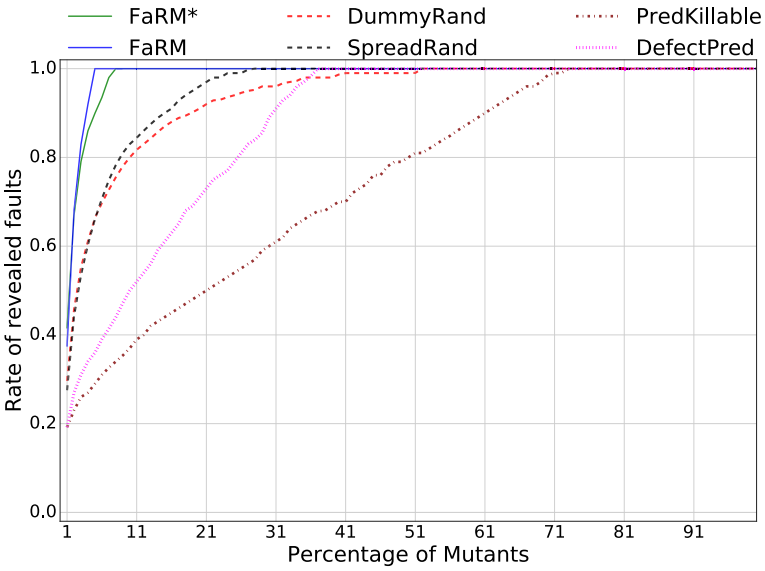


Fig. 24 Mutant prioritization performance in terms of faults revealed (median case) on CodeFlaws. The x-axis represent the number of considered mutants. The y-axis represent the ratio of the fault revealed by the strategies

mutants recorded in x-axis. From this graph we can see that all approaches reach their maximum median mutation score value when considering more than 30% of the mutants. This implies that the benefits are reduced for every approach that consider more than 30% of the involved mutants (Figs. 25 and 26).

Interestingly, both Figs. 27 and 28 demonstrate that *FaRM* guides the mutant selection towards mutants that do not maximize the mutation score nor the subsuming mutation score (random mutant selection achieves higher mutation and subsuming mutation scores than *FaRM*). Instead the selected mutants maximize fault revelation as demonstrated in Figs. 25 and 26.

Given that a large proportion of the mutants are not killable (Fig. 27), we present in Fig. 29 the sensitivity of the approaches with regard to the equivalent mutants, to see how they are ranked. We observe that *PredKillable* does quite well at ranking the killable mutants first, and *FaRM** inherit of such characteristic from *FaRM** relatively well. We also observe that *FaRM* tend to keep equivalent mutants away from the top ranks.

To provide a general view of the fault revelation trend, Figs. 25 and 26 illustrate the overall (median) effectiveness of the mutant prioritization by *FaRM* in comparison with random strategies for the ratios of selected mutants from 1% to 10%. We note that for all percentage of mutants, *FaRM* outperforms random-based prioritization. The performance improvement goes from 0% to 10% of more faults revealed when 5% and 2% of mutants are killed. These trends are similar with those we observe on CodeFlaws, suggesting that *FaRM* effectively learns the properties of the important mutants.

Required Tests Cost Metric Figure 30 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the five approaches under evaluation. While both *FaRM* and *FaRM** yield an APFD median of 92%, and *PredKillable* yields an

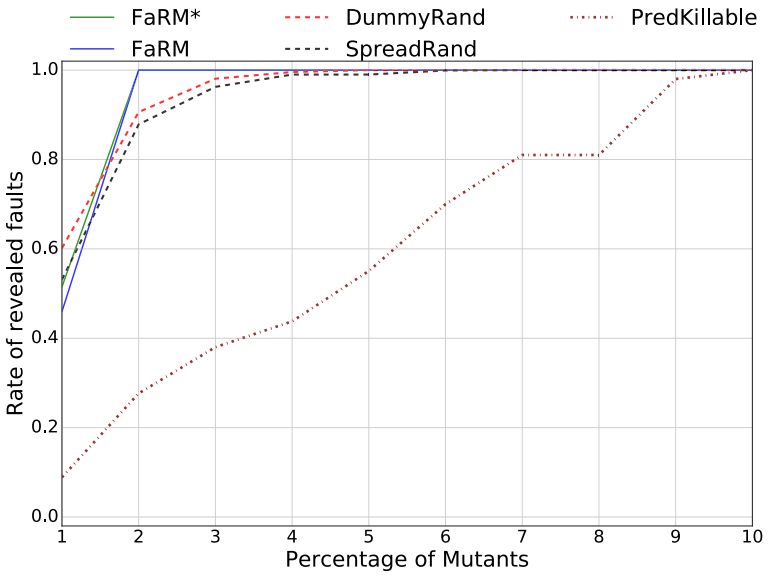


Fig. 25 *FaRM* performance in terms of faults revealed (median case) on CoREBench considering all mutants. The x-axis represent the number of considered mutants, while the y-axis represent the ratio of the fault revealed by the strategies

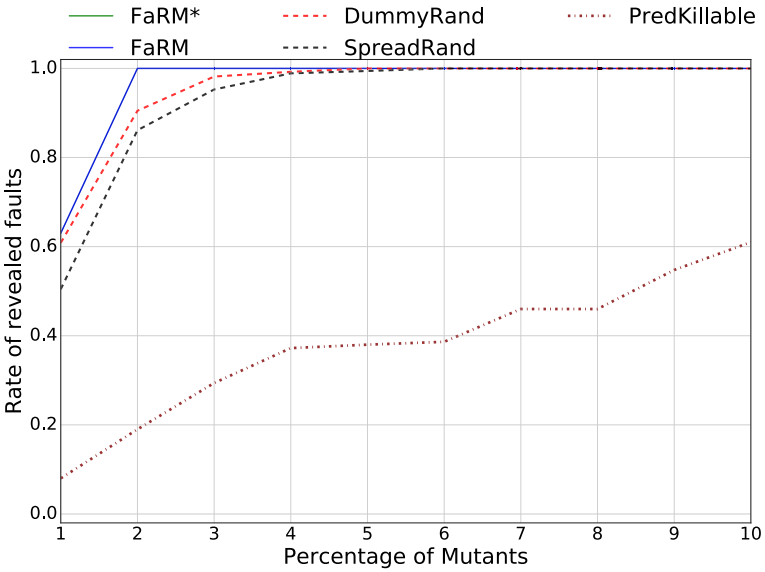


Fig. 26 *FaRM* performance in terms of faults revealed (median case) on CoREBench considering only killable mutants. The x-axis represent the number of considered mutants, while the y-axis represent the ratio of the fault revealed by the strategies

APFD median of 79%, DummyRandom and SpreadRandom reach median APFD values of 83% and 81% respectively. These results reveal that the general trend is in favour to our approach. As our approaches *FaRM* and *FaRM** are better than the random baseline,

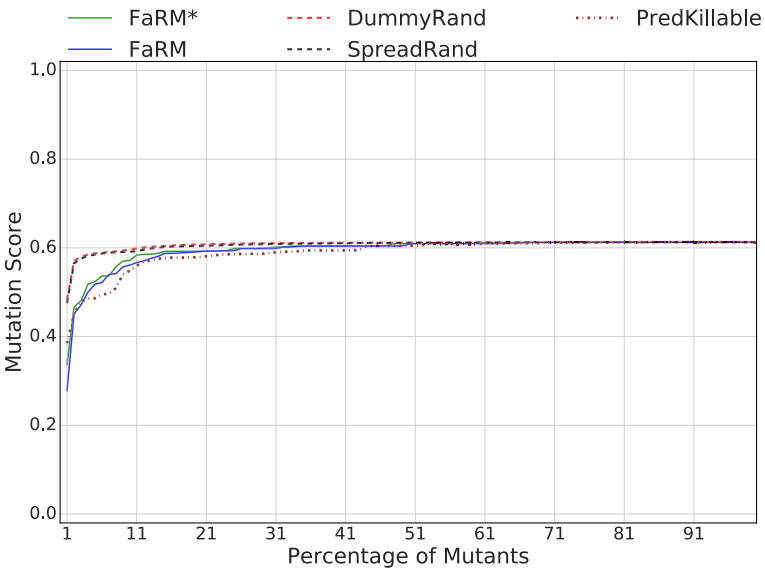


Fig. 27 Mutation score (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the mutation score attained by the strategies

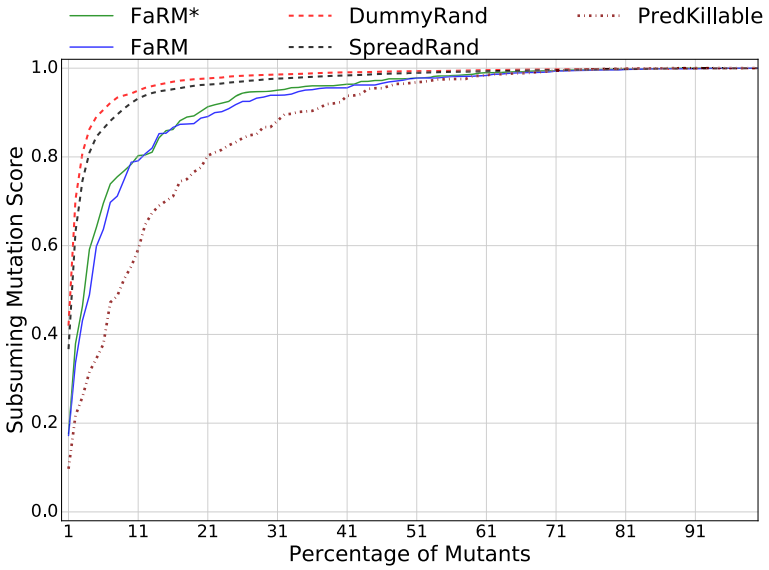


Fig. 28 Subsuming Mutation score (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the subsuming mutation score attained by the strategies

when the main cost factor (number of test that need to be designed and executed) is aligned, we can infer that it is generally better with practically important differences (of 9%). The *PredKillable* performs slightly worse than the random baseline.

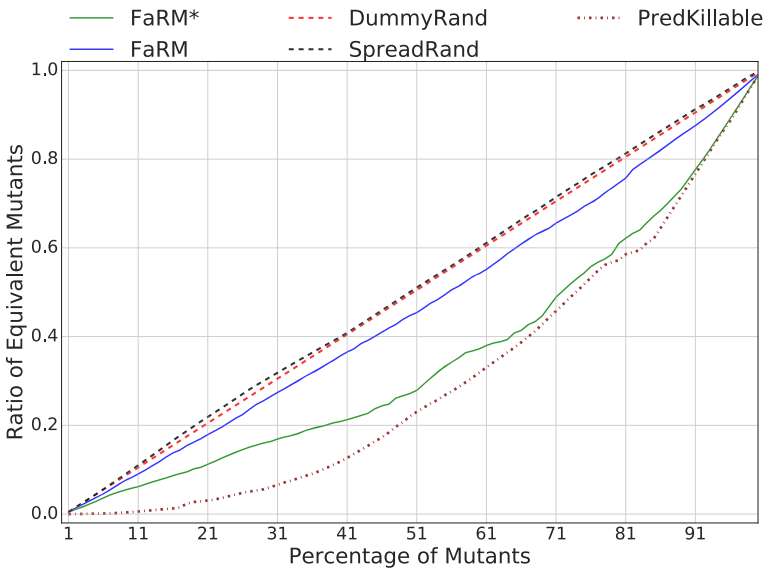


Fig. 29 Ratio of equivalents (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the proportion of equivalent mutants selected by the strategies

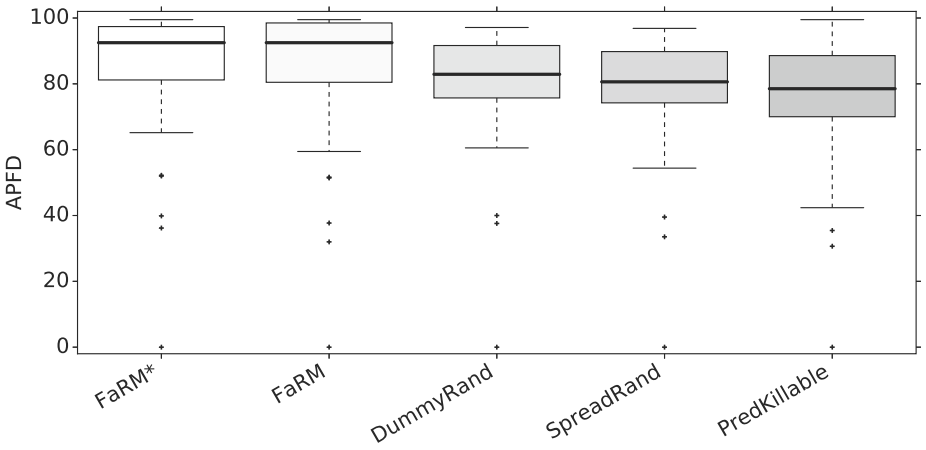


Fig. 30 APFD measurements on CoREBench for the required tests cost metric. The *FaRM* prioritization outperform the random baselines

The results of the Vargha Delaney $\hat{A}_{12}$ effect size show that *FaRM* is better than DummyRandom, SpreadRandom and *PredKillable* in 74%, 77% and 86% of the cases respectively. *FaRM** is better than DummyRandom, SpreadRandom and *PredKillable* in 70%, 74% and 81% of the cases respectively. *PredKillable* is worse than the DummyRandom and SpreadRandom in 70% and 66% of the cases respectively.

To provide a general view of the trends, Fig. 31 illustrates the overall (median) effectiveness of the required test prioritization by *FaRM* , *FaRM** and *PredKillable* in comparison

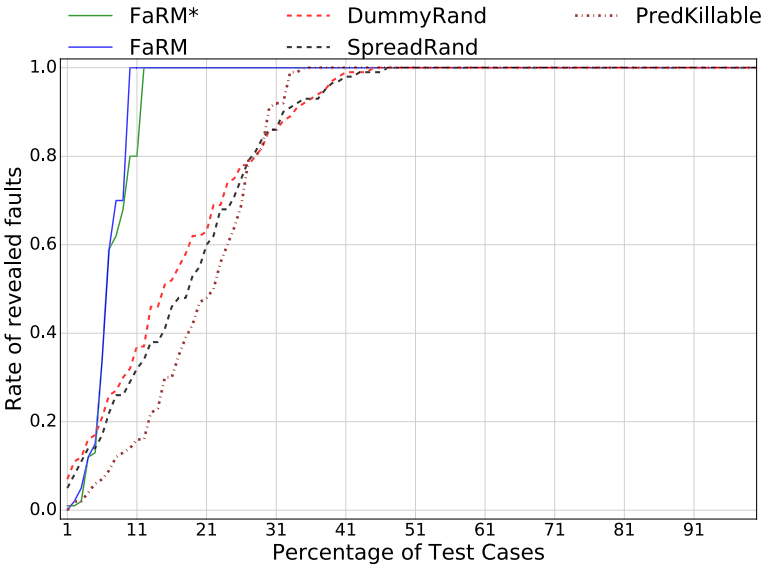


Fig. 31 Required tests prioritization performance in terms of faults revealed (median case) on CoREBench. The x-axis represent the number of considered tests. The y-axis represent the ratio of the fault revealed by the strategies

with random strategies. We note that *FaRM* and *FaRM** outperforms random-based prioritization while *PredKillable* is outperformed by the random-based prioritization. Overall, we observe that the fault revelation benefit of *FaRM* over the random approaches is above 30% (maximum difference is 70%) for the 5% to 20% top ranked tests.

Analysed Mutants Cost Metric The results of the Vargha Delaney $\hat{A}_{12}$ effect size values related to the analysed mutants cost metric show that *FaRM* and *FaRM** are better than DummyRandom and SpreadRandom. *FaRM* is better than DummyRandom and SpreadRandom in 58% and 60% of the cases respectively. The performance difference is higher for *FaRM** where it is better than DummyRandom and SpreadRandom in 61% and 63% of the cases respectively. *PredKillable* is better than DummyRandom and SpreadRandom in 56% and 58% of the cases respectively.

*FaRM** shows a larger improvement than *FaRM* over the random baseline.

Taken together our results demonstrate that *FaRM* and *FaRM** achieves significant improvements over the random baselines on both CodeFlaws and CoREBench fault sets. Therefore, the improvements made by *FaRM* and *FaRM** can be considered as important.

7 Discussion

7.1 Working Assumptions

Our approach uses machine learning to support mutation testing. As such it makes some assumptions that should hold in order to be applicable and effective. First, we assume that there are sufficient historical data from applications of the same context or previous software releases. This means that we need to have a diverse and comprehensive set of defects where mutation testing has been applied. Of course these defects need to belong to the targeted, by the testing procedure, class of defects. In the absence of sufficient defects, we can relax this requirement by training on hard-to-kill or subsuming mutants. This can be easily performed, the same way we train for equivalent mutants, as long as we have a large codebase that is sufficiently tested.

Second, we assume that defect causes are repeated. This is an important assumption as in its absence machine learning cannot work. We believe that it holds given the evidence provided by the n-version programming studies (Leveson 1995; Knight and Leveson 1986) and the empirical observations in the context of Linux kernel (Palix et al. 2011).

Third, we assume that mutants are linked with targeted defects. This assumption comes with the use of mutation testing. We believe that it holds given the empirical evidence provided by recent studies (Titcheu Chekam et al. 2017; Petrovic and Ivankovic 2018; Ramler et al. 2017; Papadakis et al. 2018b; Just et al. 2014b). Finally, we assume that fault revelation utility can be captured by static features such as the ones used in this study. We are confident that this assumption holds given the reports of Petrovic and Ivankovic (Petrovic and Ivankovic 2018) on the utility of the AST features in mutant selection and the evidence we provide here.

7.2 Threats to Validity

We acknowledge the following threats that could have affected the validity of our results. One possible external validity threat lies in the nature of the test subjects we used. Individually, the majority of programs in comparison experiments are small in size, and may not

be representative of real-world programs. Our mitigation strategy is discussed in the following subsection (Section 7.3). Moreover, since the properties of the fault revealing mutants reside on the code parts that are control and data dependent to and from the faults, the cumulative size of relevant code parts (based on which we get the feature values) should be small. Therefore, for such a study, the most important characteristics should be the faulty code area and its dependencies. Since we have a large and diverse set of real faults, we feel that this threat is limited. Future work should validate our findings and analysis to larger programs.

Another potential threat relates to the mutation operators we used. Although we have considered a variety of operators, we cannot guarantee that they yield representative mutants. To diminish this threat we used a large number of operators (816 simple operators across 18 categorises) covering the most frequently used C features. We also included all the operators adopted by the modern mutation testing tools (Titcheu Chekam et al. 2017; Papadakis et al. 2018a).

Threats to internal validity lie in the use of recent machine learning algorithms to the detriment of established and widely used techniques. Nevertheless, these threats are minimized as gradient boosting is gaining a momentum in the research literature as well as the practice of machine learning.

Similarly, there might be some issues related to code redundancy, duplicated code, that may influence our results. We discuss our redundancy mitigation strategy on Section 7.4.

Another internal validity threat may be due to the features we use. These have not been optimized with any feature selection technique. This is not a big issue in our case as we use gradient boosting that automatically performs feature selection. To verify this point we trained a Deep Learning model that also performs feature selection and checked its performance. The result showed insignificant differences from our method. Additionally, we retrained our classifiers using the features with information gain greater or equal to 0.02 and got results similar to random, suggesting that all our features are needed. Future research should seed light on this aspect by complementing and optimizing our feature set.

Other internal validity threats are due to the way we treated mutants as equivalent. To deal with this issue, we used KLEE, a state of the art test generation tool and the accompanied test suites. As the programs we are using are small KLEE should not have a problem at generating effective test suites. Together these tools kill 87% of all the mutants, demonstrating that our test suites are indeed strong. Since the 13% of the mutants we treat as equivalent is in line with the results reported by the literature (Papadakis et al. 2015), we believe that this threat is not important. Unfortunately, we cannot practically do much more than that, as the problem is undecidable (Budd and Angluin 1982).

Finally, our assessment metrics may involve some threats to construct validity. Our cost measurement, number of selected, analysed mutants and number of test cases essentially captures the manual effort involved. Automated tools may reduce this cost and hence influence our measurements. Regarding equivalent mutants, we used a state-of-the-art equivalent mutant detection technique, TCE (Papadakis et al. 2015), to remove all trivially equivalent mutants before conducting any experiment. Therefore, the remaining equivalent mutants are those that remain undetectable by the current standards. Regarding the test generation cost, we acknowledge that while automated tools manage to generate test inputs, they fail generating test oracles. Therefore, augmenting the test inputs with test oracles, remains a manual activity, which we approximate by measuring the number of tests. In our experiments we bypassed the oracle problem by using the 'correct' program versions as oracles. An alternative scenario involves the use of automated oracles, but these are rare in practice

and we did not considered them. Overall, we believe that with the current standards, our cost measurements approximate well the human cost involved.

All in all, we aimed at minimizing any potential threats by using various comparisons scenarios, clearly evaluating the benefit of the different steps in *FaRM* , and leveraging frequently used and established metrics. Additionally, to enable replication and future research we make our data publicly available[5].

7.3 Representativeness of Test Subjects

Most of our results are based on Codeflaws. We used this benchmark because machine learning requires lots of data and Codeflaws is, currently, the largest benchmark of real faults on C programs. Also because of its manageable size, we can automatically generate a relatively large and thorough test pool and apply mutation testing. Still this required 8,009 CPU days of computations (only for the mutant executions), indicating that we reach the experimentally achievable limits. Similarly, applying mutation testing on the 45 faults on CoREBench required 454 CPU days of computations.

The obvious differences between the size of the test subjects raise the question of whether our conclusions hold on other programs and faults. Fortunately, as already discussed our results on CoREBench have similar trends with those observed on Codeflaws. Training a classifier on CoREBench yields AUC values around 0.616, which is approximately the same (slightly lower) than the one we get from Codeflaws. This fact provides confidence that our features do capture the mutant properties we are seeking for. To further cater for this issue, we also selected the harder to reveal faults (faults revealed by less than 25% of the tests). This is a quality control practice, used in fault injection studies, ensures that our faults are not trivial.

Additionally, we checked the syntactic distance of the Codeflaws faults and show that it is small (please refer to Fig. 7), similarly to the one assumed by mutation testing. This property together with the subtle faults (faults revealed by less than 25% of the tests) we select make our fault set compliant with the mutation testing assumptions, i.e., the Competent Programmer Hypothesis.

Furthermore, we computed and contrasted the correlation between mutants and faults on three defect benchmarks; CoREBench, Codeflaws and Defects4J dataset (Just et al. 2014a). Our aim is to check whether there are major difference in the relation between the faults and mutants of the three benchmarks.

Defects4J is a popular defect dataset for Java, with real faults from large open source programs. To compute the correlations on Defects4J we used the data from the study of Papadakis et al. (2018b), while for CoREBench and Codeflaws we used the data from this paper. We computed the Kendall correlations with uncontrolled test suite size between 1% and 15% of all tests. We make 10,000 random test sets each with size randomly chosen between 1% and 15% of all the tests. Then, we compute the mutation score and the fault revelation of each test set, and compute the Kendall correlation between the mutation score and fault revelation. Figure 32 shows the correlations for Codeflaws, CoReBEnch and Defects4J. As can be seen, the correlations are similar in all three cases. Therefore, since the mutants and faults relations share similar properties on all cases, we believe that our defect set provide good indications on the fault revealing ability of our approach.

[5] https://mutationtesting.uni.lu/farm

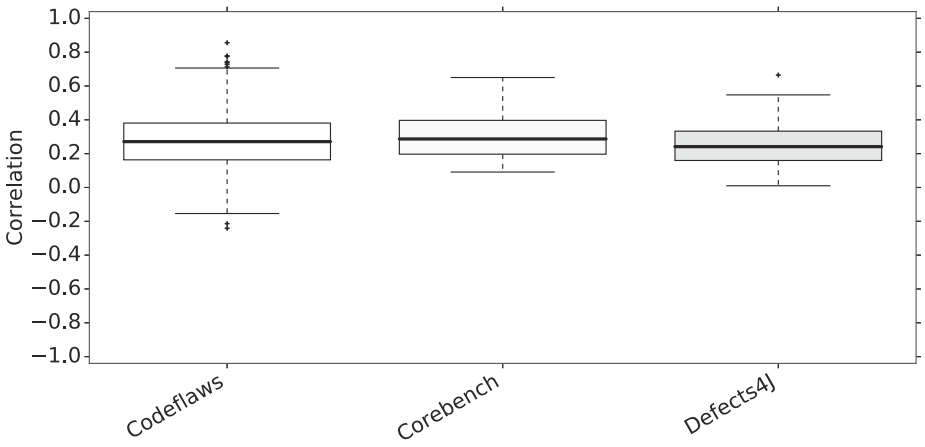


Fig. 32 Correlations between mutants and faults in three defect datasets. Similar correlations are observed in all three cases suggesting that Codeflaws provides good indications on the fault revealing ability of the mutants

7.4 Redundancy Between the Considered Faults

Code redundancy may influence our results. As depicted in Fig. 6, in Codeflaws the number of implementations for the same problem is usually higher than one. This introduces a risk that our evaluation, test defect set, may benefit from the knowledge gained during training, in case there is another implementation for the same problem in this set. Although such a case is unlikely as all of our defects are different and form unique program versions, to remove any threat from such a factor we repeated our experiment by randomly splitting the Codeflaws subjects into training and test sets in such a way that all implementations of the same problem either appear in the training set or in the evaluation, but not both. We obtained almost identical results with the previous experiment, i.e., we get AUC values of 62% and 87% respectively for fault revelation and equivalence prediction when controlling for the implementations (having always different problem implementations on the training and test sets). Another threat related to code redundancy may have affected our results in CoREBench. Among the 45 CoREBench faults we consider, only 20 of them are on the same components (13 in Coreutils, 3 in Find, and 4 in Grep). Note that the 13 instances of Coreutils form 3 separated set of 2, 5 and 6 bugs on the same component. We manually checked these defects and found that they all differ (they are located in different code parts and the code around the locations modified to fix the defects differs). Nevertheless, still there is a possibility that code similarities may impact positively or negatively our classifiers. Although such a case is compatible with our working scenario (we assume that we have similar historical data), so it is not a problem for our approach, it is still interesting to check the classifier performance on similar/dissimilar implementations.

To deal with this case, we divided our fault set (CoREBench) into two sets, one with the faults having similar faulty functions and one with dissimilar ones. To do so, we used the Deckard (Jiang et al. 2007) tool, which computes the similarity between code instances (at the AST level). For each faulty function, the tool compares the vector representation of the sub-trees of small code snippets and reports similarity scores. Two codes are considered as

similar if they have code parts with high similarity scores on the utilized abstraction, i.e., above 95% (Jiang et al. 2007).

Having divided the fault sets as similar and dissimilar we then contrast the results they provide. Overall, we found insignificant differences between the two sets. Figure 33 compares the ranking position of the fault revealing mutants in the order provided by *FaRM* , when using the following tool parameters: similarity threshold 95, 4 strides and 50 minimum number of tokens. From these results we see that there are no significant differences between the two sets, suggesting that code redundancy does not affect our results.

In order to further reduce the threat related to code redundancy, concerning mutants that appears on same line of code in the training data and test data, we repeated the experiments by removing mutants in the test data that could cause this threat. In fact, we removed all the mutants of the test data for which there exist at least one mutant in the training data, located on the same component, that have the same features. This procedure led to removing 13% (median case) of the test data mutants. The evaluation on the remaining mutants (test data after dropping the "duplicated") resulted in similar results as not dropping those mutants, i.e., AUC value of 62%. It is noted that 18% (median case) of the dropped mutants have different fault revelation score with their "duplicates" in training set. This has the unfortunate effect of confusing the classifier.

7.5 Other Attempts

Our study demonstrates how simple machine learning approaches can help improving mutation testing. Since our goal was to demonstrate the benefits of using such an approach we did not attempted to manipulate our data in any way (apart from the exclusion of the trivial faults). We achieve this goal, but still there is room for improvement that future research can exploit. For instance it is likely that classification results can be improved by pre-processing training data, e.g., exclude fault types that are problematic (Papadakis et al. 2018c), excluding fault types with few instances, excluding versions with low strength test suites, as well as by removing many other sources of noise.

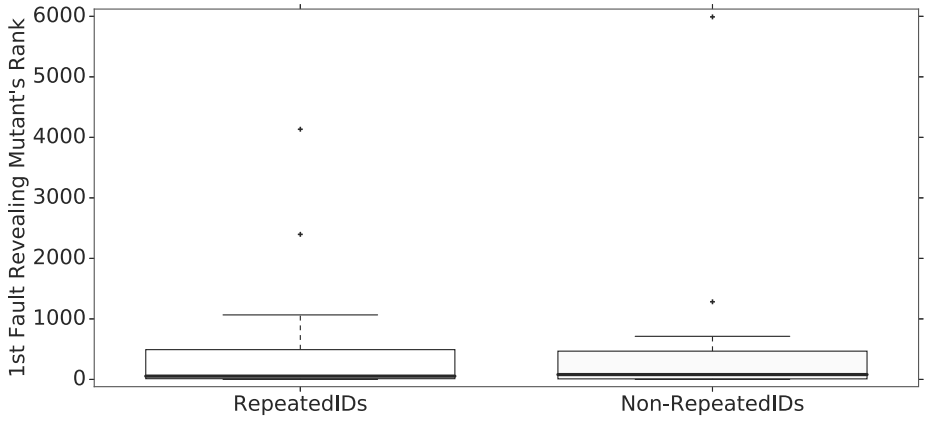


Fig. 33 CoREBench results on similar (repeatedIDs) and dissimilar (Non-RepeatedIDs) implementation. We observe similar trend in both cases suggesting a minor or no influence of code similarity on *FaRM* performance

Data manipulation strategies we attempted during our study were oversampling, the exclusive use of features with high information gain, the use of a Deep Learning classifier and targeting irrelevant mutants (the mutants with lowest fault revealing probability). Oversampling consist of randomly duplicating the data items of the minority class in order to have a more balanced data to train the classifier. In this case, we applied oversampling of the minority class for mutants which is the fault revealing class (they represent approximately 3% of the whole data). We also attempted to replace the supervised learning algorithm used by our approach by substituting the decision tree with a deep neural network classifier. We also retrained the classifier to target irrelevant mutant (mutant not killed by fault revealing tests), the motivation being that the classifier may perform better to separate irrelevant mutants than fault revealing ones. All these attempts yielded quite similar or worse results with those we report and thus, we do not detail them.

In another attempt, we trained our classifier by only using the features that have highest information gain (those with $IG \geq 0.02$ in Fig. 10) but achieved results similar to random mutant selection.

8 Related Work

Years of research in mutation testing has shown that designing tests that are capable of revealing mutant-faults results in strong test suites that in turn reveal real faults (Frankl et al. 1997; Li et al. 2009; Titcheu Chekam et al. 2017; Papadakis et al. 2018b; Just et al. 2014b). The technique is particularly effective and capable of revealing more faults than most of the other structural test criteria (Frankl et al. 1997; Li et al. 2009; Titcheu Chekam et al. 2017). Experiments using real faults have shown that mutation testing reveals more faults than the all-uses test criterion et al. (Frankl et al. 1997), and also that it reveals significantly more faults than the statement, branch and weak mutation (Titcheu Chekam et al. 2017) test criteria.

Although effective, mutation requires too many mutants making the cost of generating, analysis and executing them particularly high. Recent studies have shown that only a small number of mutants is sufficient to represent them (Kintis et al. 2010; Ammann et al. 2014; Papadakis et al. 2016) and that the majority of the mutants are somehow "irrelevant" to the underlying faults (faults that testers seek for) (Papadakis et al. 2018b). Along these lines, Natella et al. (2013) experimented with fault injection and demonstrated that up to 72% of injected faults are non representative. Papadakis et al. (2018c) analysed different types of mutants, i.e., hard to kill, subsuming, hard to propagate and fault revealing, and demonstrated that the class of fault revealing mutants is unique and differs from the other mutant sets. These studies motivated our research by indicating that it is possible to target a specific (small) set of mutants that maximize testing effectiveness.

Since the early days of mutation testing, researchers realised that the number of mutants is one of the most important problems of the method. Therefore, several approaches have been proposed to address this problem. Mutant random sampling was one of the first attempts (Budd 1980; Acree 1980). Random sampling was evaluated by Wong (1993) who found that a sampling ratio of 10% results in a test effectiveness loss of approximately 16% (evaluated on Fortran programs using the Mothra mutation testing system (DeMillo et al. 1988)). More recently, Papadakis and Malevris (2010b), using the Proteum mutation testing tool (Delamaro et al. 2001), reported a fault loss on C operators of approximately 26%, 16%, 13%, 10%, 7% and 6% for sampling ratios of 10%, 20% ..., 60% respectively.

An alternative approach to reduce the number of mutants is to select them based on their types, i.e., according to the mutation operators. Mathur (1991) introduced the idea of constrained mutation (also called selective mutation), using only two mutation operators. Wong and Mathur (1995a) experimented with sets of operators and found that two operators alone have a test effectiveness loss of approximately 5%. Offutt et al. (1993, 1996a) extended this idea and proposed a set of 5 operators, which had almost no loss on its test effectiveness. This 5 mutation operator set is considered as the current standard of mutation as it has been adopted by most of the modern mutation testing tools and used in most of the recent studies (Papadakis et al. 2018a).

Many additional selective mutation approaches have been proposed. Mresa and Bottaci (1999) defined a selective mutation procedure focused on reducing the number of equivalent mutants, instead of the number of mutants alone, as done by the studies of Mathur (1991) and Offutt et al. (1993, 1996a). They report significant reductions on the numbers of equivalent mutants produced by the selected operators, with marginal effectiveness loss (evaluated on Fortran with Mothra). Later, Barbosa et al. (2001) defined a selective mutation procedure aimed at reducing the computational cost of mutation testing of C programs. They found that a set of 10 operators could give almost the same results with the whole set of C operators supported by Proteum (78 operators). Namin et al. (2008) used regression analysis techniques and found that a set of 13 mutation operators of Proteum could provide substantial cost execution savings without any significant effectiveness loss (mutant reductions of approximately 93% are reported.

More recently, researchers have experimented with mutations involving only mutants deletion (Untch 2009). Deng et al. (2013) experimented with Java programs and the MuJava mutation operators (Ma et al. 2006) and reported reductions of 80% on the number of mutants with marginal effectiveness losses. Delamaro et al. (2014) defined deletion operators for C and reported that they significantly reduce the number of equivalent mutants, with again marginal effectiveness losses.

Other attempts have explored the identification of the program locations to be mutated. The key argument in these research directions is that program location is among the most important factor that determines the utility of the mutants. Sun et al. (2017) suggested selecting mutants that are diverse in terms of static control flow graph paths that cover them. Gong et al. (2017) used code dominator analysis in order to select mutants that, when they are covered, maximize the coverage of other mutants. This work applies weak mutation and attempts to identify dominance relations between the mutants in a static way.

Petrovic and Ivankovic (2018) identified the arid nodes (special AST nodes) as a source of information related to utility of the mutants. Their work uses dynamic analysis (test execution) combined with static analysis (based on AST) in order to identify mutants that are helpful during code reviews. We include such features in our study with the hope that they can also capture the properties of fault revealing mutants. Nevertheless, still as part of future work it is interesting to see how our features can fit within the objectives of code reviews (Petrovic and Ivankovic 2018).

Mirshokraie et al. (2015) used static (complexity) and dynamic (number of executions) analysis features to select mutants, for JavaScript programs, that reside on code parts that have low failed error propagation (they are likely to propagate to the program output). Their results show that more than 93% of the selected mutants are killable, and that more than 75% of the non-trivial mutants resided in the top 30% ranked code parts.

After several years of development of various selective mutation approaches, recent research has established that literature approaches perform similarly to random mutant sampling. Zhang et al. (2010) compared random mutant selection and selective mutation (using

C programs and the Proteum mutation operators) and found that there are no significant differences between the two approaches. The most recent approach is that of Kurtz et al. (2016) (using C programs and the Proteum mutation operators), which also reached the same conclusion (reporting that mutant reduction approaches, both selective mutation and random sampling, perform similarly).

From the above discussion it should be clear that despite the plethora of the selective mutation testing approaches, random sampling remains one of the most effective ones. This motivated our work, which used machine learning techniques and source code features in order to effectively tackle the problem. Moreover, as most of the methods use only one features, the mutant type, which according to our information gain results does not have relatively good prediction power, they should perform poorly. More importantly, our approach differs from the previous work in the evaluation metrics used. All previous work measured test effectiveness in terms of artificial faults (i.e., mutant kills or seeded faults found), while we used real faults. We believe that this is an important difference as our target (dependent variable) is the actual measurement of interest, i.e., the real fault revelation, and not a proxy, i.e., the number of mutants killed.

The closest studies to ours are the "predictive mutation", by Zhang et al. (2016, 2018) and Mao et al. (2019), and the "fault representativeness" of software fault injection by Natella et al. (2013). Predictive mutation testing attempts to predict the mutants killed for a given test suite without any mutant execution. It employs a classification model using both static and dynamic features (both on test suite and the mutants) and achieves remarkable results with an overall 10% error on the predicted mutation scores. Predictive mutation has a similar goal with our killable mutant prediction method. Though, predictive mutation assumes the existence of test suites, while our killable mutant prediction method does not. Nevertheless, our method targets a different problem, the prediction and prioritization of the important mutants prior to any test execution. To do so, we use only static features (on the code under test), while predictive mutation heavily relies on test code and dynamic features (Mao et al. 2019), and evaluate our approach using real faults (instead of mutants).

Natella et al. (2013) proposed removing injected faults to achieve meaningful 'representative' results and reduce the application cost of fault injection. This was achieved by employing classification algorithms that use complexity metrics. This approach has a similar goal with our fault revealing mutant selection, but in a different context, i.e., it targets emulating fault behaviour and not fault revelation. Nevertheless, Natella et al. rely on complexity metrics, which in our case do not seem to be adequate (as we show in RQ6). Still it is interesting to see how our approach performs in the fault injection context.

Another similar line of work is Evolutionary Mutation Testing (EMT) (Delgado-Pérez and Medina-Bulo 2018). EMT is a technique that attempts to select useful mutants based dynamic features (test execution traces) and uses them to support test augmentation. EMT learns the most interesting mutation operators and locations in the code under analysis using a search algorithm and mutant execution results. Overall, EMT achieve a 45% reduction on the number of mutants. Although EMT aims at the typical mutant reduction problem (while we aim at the fault revealing one), it can complement our method. Since EMT performs mutant selections after the mutant-test executions, *FaRM* can provide a much better starting point. Another way to combine the two techniques is to use the search engine of EMT, together with our features, to refine the mutant rankings.

A different way to reduce the mutants' number is to rank the live mutants according to their importance, so that testers can apply customised analysis according to their avail-

able budget. Along these lines, Schuler and Zeller (2013) used the mutants' impact to rank mutants according to their likelihood of being killable. Namin et al. (2015) introduced the MuRanker approach. MuRanker uses three features: the differences that mutants introduce (a) on the control-flow-graph representation (Hamming distance between the graphs), (b) on the Jimple representation (Hamming distance between the Jimple codes) and (c) on the code coverage differences produced by a given set of test cases (Hamming distance between the traces of the programs). Although our mutant prioritization scheme is similar to these approaches, we target a different problem, the static detection of valuable mutants. Thus, we do not assume the existence of test suites and mutants executions. The benefit of not making any such assumptions is that we can reduce the number of mutants to be analysed by testers, to be generated and executed by mutation testing tools.

9 Conclusions

The large number of mutants involved in mutation testing has long been identified as a barrier to the practical application of the method. Unfortunately, the problem of mutant reduction remains open, despite significant efforts within the community. To tackle this issue, we introduce a new perspective of the problem: the fault revelation mutant selection. We claim that valuable mutants are the ones which are most likely to reveal real faults, and we conjecture that standard machine learning techniques can help in their selection. In view of this, we have demonstrated that some simple 'static' program features capture the important properties of the fault revealing mutants, resulting in uncovering significantly more faults (6%-34%) than randomly selected mutants.

Our work forms a first step towards tackling the fault revelation mutant selection with the use of machine learning. As such, we expect that future research will extend and improve our results by building more sophisticated techniques, augmenting and optimizing the feature set, by using different and potentially better classifiers, and by targeting specific fault types. To support such attempts we make our subjects (programs & tests), feature, kill and fault revelation matrices publicly available.

Acknowledgments Thierry Titcheu Chekam is supported by the AFR PhD Grant of the National Research Fund, Luxembourg. Mike Papadakis is funded by the CORE Grant of National Research Fund, Luxembourg, (C17/IS/11686509/CODEMATES).

References

Acree AT (1980) On mutation. phdthesis Georgia Institute of Technology, Atlanta

Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press, Cambridge

Ammann P, Delamaro ME, Offutt J (2014) Establishing theoretical minimal sets of mutants. In: Seventh IEEE international conference on software testing, verification and validation, ICST 2014, Cleveland, pp 21–30. https://doi.org/10.1109/ICST.2014.13

Barbosa EF, Maldonado JC, Vincenzi AMR (2001) Toward the determination of sufficient mutant operators for c. Softw Test Verif Reliab 11(2):113–136

Böhme M, Roychoudhury A (2014) Corebench: studying complexity of regression errors. In: International symposium on software testing and analysis, ISSTA '14, San Jose, pp 105–115. https://doi.org/10.1145/2610384.2628058

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Budd TA (1980) Mutation analysis of program test data. phdthesis Yale University, New Haven

Budd TA, Angluin D (1982) Two notions of correctness and their relation to testing. Acta Inf 18(1):31–45. https://doi.org/10.1007/BF00625279

Cadar C, Dunbar D, Engler DR (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. Proceedings, San Diego, pp 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

Coles H, Laurent T, Henard C, Papadakis M, Ventresque A (2016) PIT: a practical mutation testing tool for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, pp 449–452. https://doi.org/10.1145/2931037.2948707

Delamaro ME, Maldonado JC, Vincenzi AMR (2001) Proteum/IM 2.0: An Integrated Mutation Testing Environment, Springer US, Boston, chap Mutation Testing for the New Century, pp 91–101. https://doi.org/10.1007/978-1-4757-5939-6_17

Delamaro ME, Offutt J, Ammann P (2014) Designing deletion mutation operators. In: Seventh IEEE international conference on software testing, verification and validation, ICST 2014, Cleveland, pp 11–20. https://doi.org/10.1109/ICST.2014.12

Delgado-Pérez P, Medina-Bulo I (2018) Search-based mutant selection for efficient test suite improvement: Evaluation and results. Inf Softw Technolo 104:130 – 143. https://doi.org/10.1016/j.infsof.2018.07.011, http://www.sciencedirect.com/science/article/pii/S0950584918301551

DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practicing programmer. IEEE Comput 11(4):34–41. https://doi.org/10.1109/C-M.1978.218136

DeMillo RA, Guindi DS, King KN, McCracken WM, Offutt AJ (1988) An extended overview of the mothra software testing environment. In: Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88), Banff Alberta, pp 142–151

DeMillo RA, Offutt AJ (1991) Constraint-based automatic test data generation. IEEE Trans Softw Eng 17(9):900–910. https://doi.org/10.1109/32.92910

Deng L, Offutt J, Li N (2013) Empirical evaluation of the statement deletion mutation operator. In: Sixth IEEE international conference on software testing, verification and validation, ICST 2013, Luxembourg, pp 84–93. https://doi.org/10.1109/ICST.2013.20

Ferrari FC, Pizzoleto AV, Offutt J (2018) A systematic review of cost reduction techniques for mutation testing: Preliminary results. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 1–10. https://doi.org/10.1109/ICSTW.2018.00021

Frankl PG, Weiss SN, Hu C (1997) All-uses vs mutation testing: an experimental comparison of effectiveness. J Syst Softw 38(3):235–253. https://doi.org/10.1016/S0164-1212(96)00154-9

Frankl PG, Iakounenko O (1998) Further empirical studies of test effectiveness. SIGSOFT Softw Eng Notes 23(6):153–162. https://doi.org/10.1145/291252.288298

Fraser G, Zeller A (2012) Mutation-driven generation of unit tests and oracles. IEEE Trans Softw Eng 38(2):278–292. https://doi.org/10.1109/TSE.2011.93

Friedman JH (2002) Stochastic gradient boosting. Comput Stat Data Anal 38(4):367–378

Gong D, Zhang G, Yao X, Meng F (2017) Mutant reduction based on dominance relation for weak mutation testing. Inf Softw Technol 81:82–96. https://doi.org/10.1016/j.infsof.2016.05.001

Hariri F, Shi A, Converse H, Khurshid S, Marinov D (2016) Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In: 27th IEEE international symposium on software reliability engineering, ISSRE 2016, Ottawa, pp 105–115. https://doi.org/10.1109/ISSRE.2016.51

Henard C, Papadakis M, Harman M, Jia Y, Traon YL (2016) Comparing white-box and black-box test prioritization. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, pp 523–534. https://doi.org/10.1145/2884781.2884791

Jia Y, Harman M (2009) Higher order mutation testing. Inf Softw Technol 51(10):1379–1393. https://doi.org/10.1016/j.infsof.2009.04.016

Jiang L, Misherghi G, Su Z, Glondu S (2007) Deckard: Scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering (ICSE'07), pp 96–105. https://doi.org/10.1109/ICSE.2007.30

Just R, Jalali D, Ernst MD (2014a) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: International symposium on software testing and analysis, ISSTA '14, san Jose, pp 437–440. https://doi.org/10.1145/2610384.2628055

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014b) Are mutants a valid substitute for real faults in software testing?. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, 654–665. https://doi.org/10.1145/2635868.2635929

Just R, Kurtz B, Ammann P (2017) Inferring mutant utility from program context. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, pp 284–294. https://doi.org/10.1145/3092703.3092732

Kamei Y, Shihab E (2016) Defect prediction: Accomplishments and future challenges. In: Leaders of tomorrow symposium: Future of software engineering, FOSE@SANER 2016, Osaka, pp 33–45. https://doi.org/10.1109/SANER.2016.56

Keck T (2016) Fastbdt: A speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification. arXiv:160906119

Kintis M, Papadakis M, Malevris N (2010) Evaluating mutation testing alternatives: A collateral experiment. In: 17th asia pacific software engineering conference, APSEC 2010, Sydney, 300–309. https://doi.org/10.1109/APSEC.2010.42

Kintis M, Papadakis M, Jia Y, Malevris N, Traon YL, Harman M (2018) Detecting trivial mutant equivalences via compiler optimisations. IEEE Trans Softw Eng 44(4):308–333. https://doi.org/10.1109/TSE.2017.2684805

Knight JC, Leveson NG (1986) An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans Softw Eng 12(1):96–109. https://doi.org/10.1109/TSE.1986.6312924

Kurtz B, Ammann P, Offutt J, Delamaro ME, Kurtz M, Gökçe N (2016) Analyzing the validity of selective mutation with dominator mutants. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, pp 571–582. https://doi.org/10.1145/2950290.2950322

Leveson NG (1995) Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology. Addison-Wesley

Li N, Praphamontripong U, Offutt J (2009) An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage. In: Mutation 2009, Denver, pp 220–229. https://doi.org/10.1109/ICSTW.2009.30

Ma Y, Offutt J, Kwon YR (2006) Mujava: a mutation system for java. In: 28Th international conference on software engineering (ICSE 2006), Shanghai, 827–830. https://doi.org/10.1145/1134425

Mao D, Chen L, Zhang L (2019) An extensive study on cross-project predictive mutation testing. In: 12th IEEE international conference on software testing, verification and validation, ICST 2019, Xian, pp 160–171

Mathur AP (1991) Performance, effectiveness, and reliability issues in software testing. In: Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC'79), Tokyo, pp 604–605

Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13. https://doi.org/10.1109/TSE.2007.256941

Mirshokraie S, Mesbah A, Pattabiraman K (2015) Guided mutation testing for javascript web applications. IEEE Trans Softw Eng 41(5):429–444. https://doi.org/10.1109/TSE.2014.2371458

Mresa ES, Bottaci L (1999) Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing. Verif Reliab 9(4):205–232

Namin AS, Andrews JH, Murdoch DJ (2008) Sufficient mutation operators for measuring test effectiveness. In: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, pp 351–360

Namin AS, Xue X, Rosas O, Sharma P (2015) Muranker: a mutant ranking tool. Softw Test Verif Reliab 25(5-7):572–604. https://doi.org/10.1002/stvr.1542

Natekin A, Knoll A (2013) Gradient boosting machines, a tutorial. Frontiers in neurorobotics 7

Natella R, Cotroneo D, Durães J, Madeira H (2013) On fault representativeness of software fault injection. IEEE Trans Softw Eng 39(1):80–96. https://doi.org/10.1109/TSE.2011.124

Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: Proceedings of the 15th International Conference on Software Engineering, ICSE '93. IEEE Computer Society Press, Los Alamitos, pp 100–107. http://dl.acm.org/citation.cfm?id=257572.257597

Offutt AJ, Craft WM (1994) Using compiler optimization techniques to detect equivalent mutants. Softw Test Verif Reliab 4(3):131–154. https://doi.org/10.1002/stvr.4370040303

Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996a) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118

Offutt AJ, Pan J, Tewary K, Zhang T (1996b) An experimental evaluation of data flow and mutation testing. Softw Pract Exper 26(2):165–176. https://doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K

Palix N, Thomas G, Saha S, Calvès C, Lawall JL, Muller G (2011) Faults in linux: ten years later. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, pp 305–318. https://doi.org/10.1145/1950365.1950401

Papadakis M, Malevris N (2010a) Automatic mutation test case generation via dynamic symbolic execution. In: IEEE 21St international symposium on software reliability engineering, ISSRE 2010, San Jose, 121–130. https://doi.org/10.1109/ISSRE.2010.38

Papadakis M, Malevris N (2010b) An empirical evaluation of the first and second order mutation testing strategies. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010. Workshops Proceedings, Paris, pp 90–99. https://doi.org/10.1109/ICSTW.2010.50

Papadakis M, Jia Y, Harman M, Traon YL (2015) Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 37th IEEE/ACM international conference on software engineering, ICSE 2015, Florence, pp 936–946, https://doi.org/10.1109/ICSE.2015.103

Papadakis M, Henard C, Harman M, Jia Y, Traon YL (2016) Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, pp 354–365. https://doi.org/10.1145/2931037.2931040

Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M (2018a) Mutation testing advances: An analysis and survey. Advances. in Computers

Papadakis M, Shin D, Yoo S, Bae DH (2018b) are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: Proceedings of the 40th International Conference on Software Engineering, ICSE

Papadakis M, Titcheu Chekam T, Le Traon Y (2018c) Mutant quality indicators. In: The 13th international workshop on mutation analysis, mutation

Petrovic G, Ivankovic M (2018) State of mutation testing at google. In: 40th IEEE/ACM international conference on software engineering: Software engineering in practice track, ICSE-SEIP 2018, Gothenburg, Sweden

Ramler R, Wetzlmaier T, Klammer C (2017) An empirical study on the application of mutation testing for a safety-critical industrial software system. In: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, pp 1401–1408. https://doi.org/10.1145/3019612.3019830

Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27(10):929–948. https://doi.org/10.1109/32.962562

Schuler D, Zeller A (2013) Covering and uncovering equivalent mutants. Softw Test. Verif Reliab 23(5):353–374. https://doi.org/10.1002/stvr.1473

SiR (2018) Software-artifact infrastructure repository. https://sir.unl.edu/portal/bios/tcas.php, accessed: 2018-10-20

Sun C, Xue F, Liu H, Zhang X (2017) A path-aware approach to mutant reduction in mutation testing. Inf Softw Technol 81:65–81. https://doi.org/10.1016/j.infsof.2016.02.006

T Acree A, A Budd T, Demillo R, J Lipton R, G Sayward F (1979) Mutation analysis. Technical Report GIT-ICS-79/08, pp 92

Tan SH, Yi J, Yulis Mechtaev S, Roychoudhury A (2017) Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, 2017 - Companion Volume, pp 180–182. https://doi.org/10.1109/ICSE-C.2017.76

Titcheu Chekam T, Papadakis M, Le Traon Y, Harman M (2017) An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, pp 597–608. https://doi.org/10.1109/ICSE.2017.61

Untch RH (2009) On reduced neighborhood mutation analysis using a single mutagenic operator. In: Proceedings of the 47th Annual Southeast Regional Conference, 2009, Clemson. https://doi.org/10.1145/1566445.1566540

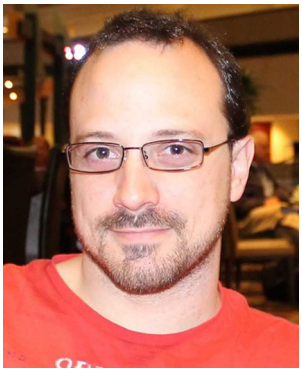
Vargha A, Delaney HD (2000) A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. Jrnl Educ Behav Stat 25(2):101–132

Wen M, Chen J, Wu R, Hao D, Cheung S (2018) Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, pp 1–11. https://doi.org/10.1145/3180155.3180233

Wong WE (1993) On mutation and data flow. phdthesis Purdue University, West Lafayette

Zhang L, Hou S, Hu J, Xie T, Mei H (2010) Is operator-based mutant selection superior to random mutant selection?. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, pp 435–444. https://doi.org/10.1145/1806799.1806863
Zhang L, Marinov D, Khurshid S (2013) Faster mutation testing inspired by test prioritization and reduction. In: International symposium on software testing and analysis, ISSTA '13, Lugano, pp 235–245. https://doi.org/10.1145/2483760.2483782
Zheng A (2015) Evaluating Machine Learning Models A Beginner's Guide to Key Concepts and Pitfalls. O'Reilly Media, Inc. https://www.oreilly.com/data/free/evaluating-machine-learning-models.csp
Wong WE, Mathur AP (1995a) Reducing the cost of mutation testing: an empirical study. J Syst Softw 31(3):185–196. https://doi.org/10.1016/0164-1212(94)00098-0
Wong WE, Mathur AP (1995b) Reducing the cost of mutation testing: an empirical study. J Syst Softw 31(3):185–196. https://doi.org/10.1016/0164-1212(94)00098-0
Zhang J, Wang Z, Zhang L, Hao D, Zang L, Cheng S, Zhang L (2016) Predictive mutation testing. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, pp 342–353. https://doi.org/10.1145/2931037.2931038
Zhang J, Zhang L, Harman M, Hao D, Jia Y, Zhang L (2018) Predictive mutation testing. IEEE Trans Softw Eng:1–1. https://doi.org/10.1109/TSE.2018.2809496

Thierry Titcheu Chekam is a Ph.D. student at the Interdisciplinary Centre for Security, Reliability and Trust, the University of Luxembourg. He received the B.Sc. degree in Computer Science and Technology from the University of Science and Technology of China in 2013, and the M.Eng. degree in Software Engineering from the School of Software, Tsinghua University in 2015. His research areas comprise software testing, mutation analysis, symbolic execution, and cloud computing/storage.



Mike Papadakis is a research scientist at the Interdisciplinary Center for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received a Ph.D. diploma in Computer Science from the Athens University of Economics and Business. He is recognised for his work on software testing and in particular in the area of mutation testing. His research interests also include static analysis, prediction modelling and search-based software engineering.

Tegawendé F. Bissyandé is a research scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) of the University of Luxembourg. He received his PhD degree in Computer Sciences from the University of Bordeaux (France) in 2013. His research interests lie in Debugging and fixing software and in Empirical studies for improving software engineering processes.

Yves Le Traon is professor at the university of Luxembourg where he leads the SERVAL (SEcurity, Reasoning and VALidation) research team. His research interests within the group include (1) innovative testing and debugging techniques, (2) Android apps security and reliability using static code analysis, machine learning techniques and, (3) model-driven engineering with a focus on IoT and CPS. His reputation in the domain of software testing is acknowledged by the community. He has been General Chair of major conferences in the domain, such as the 2013 IEEE International Conference on Software Testing, Verification and Validation (ICST), and Program Chair of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). He serves at the editorial boards of several, internationally-known journals (STVR, SoSym, IEEE Transactions on Reliability) and is author of more than 150 publications in international peer-reviewed conferences and journals.

Koushik Sen is a professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His research interest lies in Software Engineering, Programming Languages, and Formal methods. He is interested in developing software tools and methodologies that improve programmer productivity and software quality. He is best known for his work on "DART: Directed Automated Random Testing" and concolic testing. He has received a NSF CAREER Award in 2008, a Haifa Verification Conference (HVC) Award in 2009, a IFIP TC2 Manfred Paul Award for Excellence in Software: Theory and Practice in 2010, a Sloan Foundation Fellowship in 2011, a Professor R. Narasimhan Lecture Award in 2014, an Okawa Foundation Research Grant in 2015, and an ACM SIGSOFT Impact Paper Award in 2019. He has won several ACM SIGSOFT Distinguished Paper Awards. He received the C.L. and Jane W-S. Liu Award in 2004, the C. W. Gear Outstanding Graduate Award in 2005, and the David J. Kuck Outstanding Ph.D. Thesis Award in 2007, and a Distinguished Alumni Educator Award in 2014 from the UIUC Department of Computer Science. He holds a B.Tech from Indian Institute of Technology, Kanpur, and M.S. and Ph.D. in CS from University of Illinois at Urbana-Champaign.

Affiliations

Thierry Titcheu Chekam[1] · Mike Papadakis[1] · Tegawendé F. Bissyandé[1] · Yves Le Traon[1] · Koushik Sen[2]

Mike Papadakis
michail.papadakis@uni.lu

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

Yves Le Traon
yves.letraon@uni.lu

Koushik Sen
ksen@berkeley.edu

[1] SnT Centre, University of Luxembourg, Luxembourg, Luxembourg

[2] University of California, Berkeley, CA, USA